Model-Based Verification of the DMAMAC Protocol in UPPAAL

A. Ajith Kumar S.[1,2]

[1] Bergen University College, Norway
Email:{aaks}@hib.no
[2] University of Agder, Norway

Abstract. Radio communications are the main energy consumers in Wireless Sensor Networks. Medium Access Control protocols are responsible for managing radio communications thus directly affect the energy efficiency of the sensor nodes. The Dual-Mode Adaptive MAC or DMAMAC protocol is a MAC protocol proposed for process control applications. The goal of the protocol is to improve energy efficiency along with satisfying other real-time requirements. The protocols used in such control applications have to be thoroughly verified for design faults. Formal verification is a technique to verify abstract representation of protocols and real-time systems. Verification assists in increasing the trust on the design of these protocols and systems. In this report, we use Uppaal, a tool with model verification capabilities, to verify the design of the DMAMAC protocol. The tool provides traces to inconsistencies or errors if any. The protocol switches between the two operational modes that correspond to two different states of process control, namely transient and steady. This state-switch in particular is a critical function for the DMAMAC protocol, which along with other functional properties is verified in this report. We have created a time-based model in Uppaal, and successfully verified important properties of the protocol, increasing confidence in the design of the protocol.

1 Introduction

A Wireless Sensor Actuator Network (WSAN) [1] consists of sensors and actuators that use radios to send, relay, and receive information. WSANs are used in application domains ranging from process and factory automation to smart home automation, and also medical applications. The driving features resulting in increased adoption of WSAN as network solutions across multiple application domains is the reduction in cost and size of the solution. Feedback based control loops that use wired or wireless solutions are collectively known as Networked Control Systems (NCS) [2]. NCS mainly used wired communication systems, but are increasingly adopting wireless for its advantages. Wireless communications have various shortcomings thus have not been a de-facto replacement to existing wired solutions. The limitations of wireless solutions include low-bandwidth, energy efficiency, interference, and packet-loss possibilities. Energy efficiency is an important concern when the devices are battery powered. Also, since these control loops are automated, issues in the wireless communication can result in halting of operations. Thus, we need to make sure that these solutions are trustable. Wireless solutions are made up of a collection of protocols that cater for different functions. Medium Access Control (MAC) is one of the protocols that are critical to the smooth working of the entire network. MAC protocol controls the communication part thus handles the radio. Radio module is the biggest consumer of energy in wireless nodes. The Dual-Mode Adaptive MAC (DMAMAC) [3] protocol is one such MAC protocol that works towards empowering control systems with energy efficient solutions. The design of protocols can sometimes have faults in it, which might be overlooked during the design process, and are determined late during implementation testing. This increases the overall cost of

design and development. One way to detect these faults early is by model-checking the design of the protocol. In this report, we verify the DMAMAC protocol to make sure the implementation meets the requirements of the protocol, and is free of issues.

Model-checking is a popular technique for early verification of protocol design. Model-checking allows for exhaustive verification of concurrent systems. It has been applied for verification of communication protocols in particular [4,5,6,7]. Verification in the early design phase ensures the correctness in the working of the protocols. Model-checking assists in discovering design faults by exhaustively traversing through all the paths in a given model execution. These faults can then be corrected resulting in a verified version of the protocol. Model-checking tools provide with traces to possible failure states, thus assisting in solving the issues. Uppaal [8] is a verification tool-suite that is used for model checking. Apart from model-checking and verification, Uppaal also allows for simulation, which could provide useful insights to the working of the protocol. 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 [8] of the Lightweight Medium Access Control (LMAC) protocol. In this report we explore the qualitative features of the DMAMAC protocol.

For model checking, an abstract representation of the protocol to be verified is used as an input. The abstract representation consists of the functional behavior of the protocol in minimal terms. In this report, we study the correctness of the DMAMAC protocol [3]. The DMAMAC protocol was proposed for NCSs with real time and energy efficiency requirements, specifically for process control applications that fluctuate between the two states of operation: steady and transient. The protocol operates in one of the operational modes/states at a given time and switches occasionally depending on the application requirements. The DMAMAC protocol is aimed at applications that have dominant steady state, of the order of $\geq 90\%$ during the entire operation. For such applications the DMAMAC protocol consumes far less energy [3] relative to single mode protocols. Given the two-state nature of process control and dual-mode operation of the DMAMAC protocol, state-switch is one of the important properties to be verified. Also, we design some queries that indicate reachability of faulty states and verify if the model reaches those faulty states.

1.1 Related Work

Uppaal in general has been used to model and verify various communication protocol protocols [5,7,6]. Ad-hoc On-Demand Distance Vector (AODV) [5], a routing protocol was model-checked using Uppaal. In [5] the authors used an AWN (process algebra) specification to obtain an Uppaal model automatically. The goal of the verification was to explore the behavior of AODV protocol in different configurations of network topology and to correct them. The Lightweight Medium Access Control (LMAC) [6] protocol is the closest MAC protocol modeling related to our work presented in this report. The DMAMAC protocol is a new MAC protocol proposed in [3] for process control applications. The LMAC and the DMAMAC protocols are two distinct protocols with distinct goals, and differ extensively in their base features. The LMAC protocol is a self-organizing 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 detecting collisions. In the DMAMAC protocol the slot scheduling is statically done pre-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 [7], the authors have focused mainly on modeling the Chipcon CC2420 transceiver but this is related in terms of their use of a collision model and how collision is observed. We use a collision model similar to [7,6] in our report. We created a timed version of the DMAMAC protocol based on the Finite State Machine

(FSM) of the nodes and sink.

The rest of the report is organized as follows. In Sec. 2 we describe the protocol in brief. For extensive details refer to [3]. In section 3, we introduce Uppaal and the modeling features in Uppaal which assists in understanding the model of the protocol. Section 4 describes in detail the Uppaal model of the DMAMAC protocol. We partly also validate the protocol along with the description. Further validation is done in Sec. 5. The verification of the protocol across various configurations is discussed in Sec. 6. Further, we conclude and discuss possible future work in Sec. 7.

2 DMAMAC Protocol

The DMAMAC protocol [3] is a MAC protocol designed for process control applications. The DMAMAC protocol has two operational modes catering for two states of the process control applications: transient mode and steady mode. The protocol is based mostly on Time-Division Multiple Access (TDMA) for data communication and Carrier Sense Multiple Access (CSMA)-TDMA hybrid for alert message communication. The basic functioning of the protocol is based on the GinMAC protocol [9] 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 fitted with actuators, which act on the data performing a physical function. We can also have a wireless node with both sensor and actuator fitted with. 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.

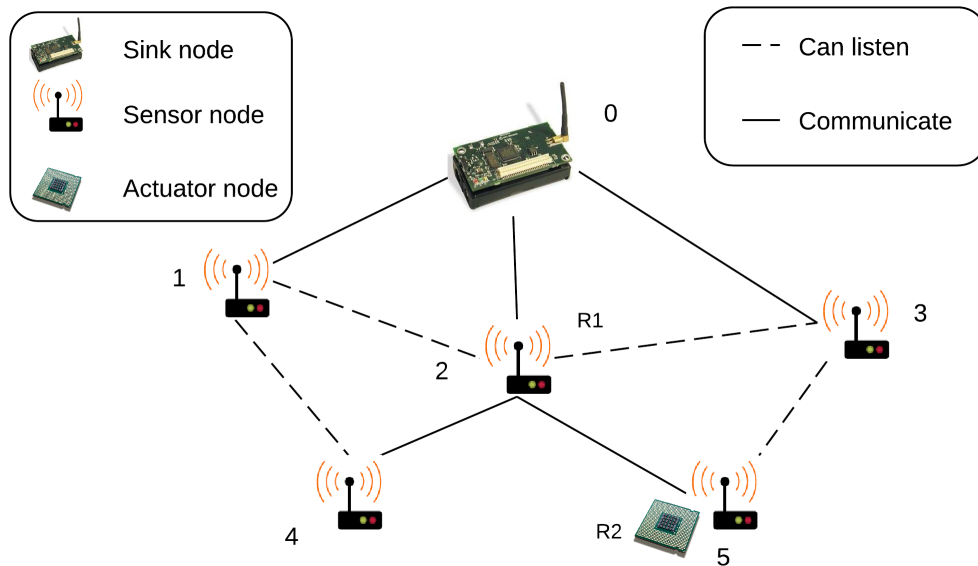

Fig. 1: The network topology for DMAMAC protocol

Similar to the GinMAC protocol, the network deployment for the DMAMAC protocol adheres to a tree topology as shown in Fig. 1. The continuous lines between nodes represent data communication. The dashed lines represent nodes which can hear each other but have no direct data communication

with (can listen to each other). Each level in the tree topology is ranked (marked with "R#"), with sink having the lowest rank number and the farthest leaf nodes having the highest rank number. This ranking is used for the alert message sending procedure. Also, similar to GinMAC we set a maximum of 25 nodes per network managed by one sink, to be able to have low data transmission delay within the network. Firstly, we discuss some assumptions that were made supporting 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, 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.
- The nodes always send data packets in their data slots.
- A static network topology is used.
- A single slot accommodates both a data packet and an acknowledgement for the same.

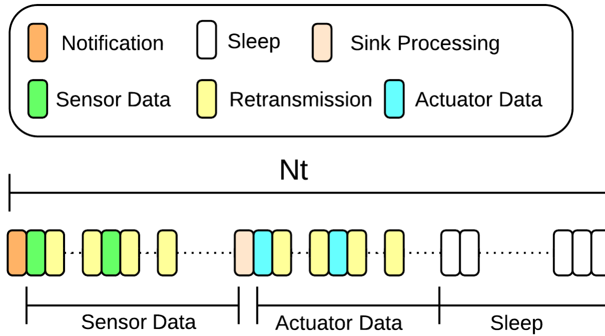

Fig. 2: The transient superframe of the DMAMAC protocol

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. 2, this is similar to the GinMAC superframe (except for changed actuator slot position). 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 from the sink to all nodes, and a sink processing slot. The sink processing slot is a symbolic slot providing an extra slot for the sink to complete all required processing, is thus not important and is skipped in the Uppaal model. 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 sensors sending data towards the sink and sink towards the actuators.
- The sleep part is executed where all sensors and actuators go to sleep in order to improve energy efficiency. (This part represents the situation where all nodes are in sleep. Individually the nodes are sleeping when they are not doing anything else).

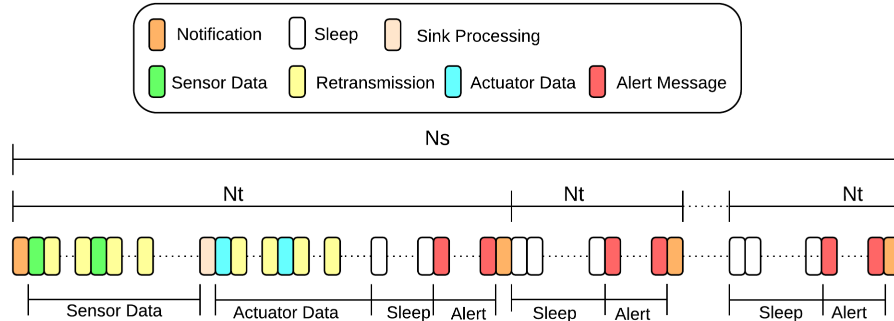

Fig. 3: The steady superframe of the DMAMAC protocol

Steady Mode The steady mode operation is designed to operate during the steady state of the process control application. In the steady state, the data requirement of the controller is relatively lower, thus the data rate is lower in the steady mode. This lower data rate makes the DMAMAC protocol energy efficient compared to the single mode protocols. The steady superframe used in the steady mode operation is shown in Fig. 3. Apart from the parts that also exist in the transient superframe, the steady superframe consists of 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 made. Note that w.r.t [3] a slightly modified steady mode superframe is used. There are notification slots placed at the end of each transient (Nt) part to facilitate immediate application of alert, and making a state-switch. This results in a lower switch delay than the previous version. In the alert part one slot is given 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 sent from the sink to all the nodes, same as the one in transient mode operation.
- The data part and the sleep part follows, similar to the working in transient mode operation.
- Lastly, the alert part starts. Sensor nodes that have alert messages to be communicated use appropriate slots provided for each rank to notify its 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 sleep 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 along with other regular notification takes place in this slot.

Change of superframes A process readily 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 is important to accommodate higher data generated 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 notification message of the sink. When the sink node switches from transient to steady, it defines a threshold interval within

which the read sensor data should lie in, and informs the sensors about this threshold interval. During the entire steady mode operation, the sensors constantly look out for 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.

Alert Message Alert message is the message created by the sensor nodes to notify the sink of a state-switch requirement. The alert slot is based on the rank of the nodes in the tree topology. The alert slots start with the farthest (from the sink) leaf nodes (highest rank) in tree topology traversing towards the sink. For sending alert messages, we use CSMA-TDMA hybrid, where each rank has a slot. The nodes having the same ranks send alert messages towards their parent with random delay combined with carrier sense. 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. During channel assessment if nodes detect another node sending an alert message it just drops its alert message. But, yet there are chances of collision when two nodes choose the same random delay, also when the two senders cannot listen to each other but the receiver listens to both. The nodes check for change of operational mode following the sending of the alert, if no change occurs (because of collision) it saves the alert and sends the alert again in the next alert slot.

2.1 Finite State Machine

We represent the working of the nodes and the sink using the DMAMAC protocol with their Finite State Machine (FSM). The FSM gives a simpler version of the Uppaal model designed for the protocol. The FSM of the nodes include the following states: *StateController*, *Notification*, *Sent* (data sent), *Received* (data received), *SendAlert*, *ReceiveAlert*, and *Sleep*. The FSM of the sink is the same as node FSM except for the *SendAlert* state. The FSM for the node model is shown in Fig. 4 and for the sink model is shown in Fig. 5. Here Tx represent send or transmission, and Rx represents receive.

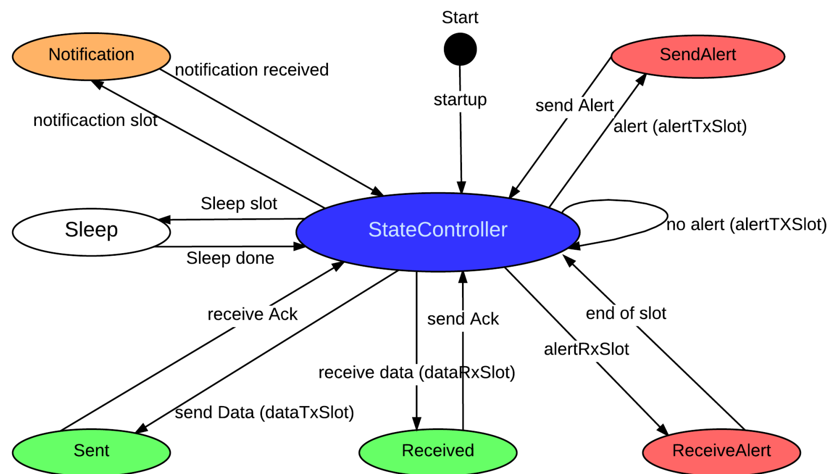

Fig. 4: FSM for the node with the DMAMAC protocol

The *StateController* represents a state where the handling of state transition occurs. The *Notification* state is reached both by sink and the nodes in the notification slot (marked as a transition), and nodes

exit from the *Notification* state when they receive a notification message, and the sink exits when it sends the notification message. When the nodes or the sink send data (in their data send slot) they reach the *Sent* state. Nodes and sink leave the *Sent* state when an acknowledgement (Ack) is received. On receiving data the nodes and the sink reach the *Received* state, and exit it by sending an Ack. Alert messages are handled differently from data and Ack, the nodes or the sink go to the *receiveAlert* state in the beginning of their receive alert slot (alertRxSlot), and stay in that state until the end of the slot. *SendAlert* state is reached when the nodes have any alert to send and are in their alert transmission slot (alertTxSlot). Nodes can have alert to be sent in two cases, when they have an alert of their own or if they have to forward alert. In a case where nodes have no alert to send, they move back to the *StateController* state shown with a self-loop edge. The alert choice when initiated by the node is actually based on probability (send or not send), and is better represented in the Uppaal model with probability weights associated with the edges.

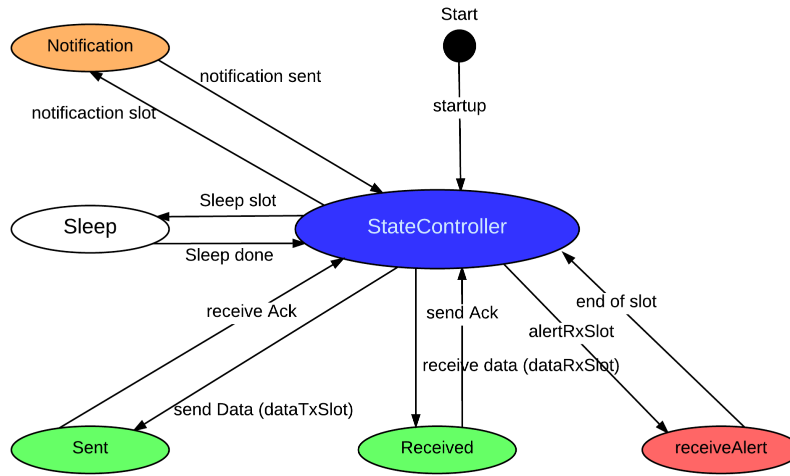

Fig. 5: FSM for the sink with the DMAMAC protocol

2.2 Properties

Important properties of the DMAMAC protocol are listed here. These properties are used to verify the correctness in the design of the protocol. We list these properties in a simple language representation here, later we transform these into queries in the Uppaal query language, and verify them. Given the dual-mode operation of the DMAMAC protocol, the important properties include operating in different modes, and switching between them. Also, note that given the two operational modes, the system can be started with either operational mode which can affect the result of certain queries. Below, we list the properties which are labeled, and used later in Sec. 5 and Sec. 6.

- No deadlock: A generic safety property defining the absence of deadlock in the protocol design.
- State 1a: There should not exist a state where the system is not in either of the operational modes. These kinds of states are called faulty states and should not exist in any execution paths of the model.
- State 1b: Another property about operational modes to conform that both sink and the nodes are always on the same operational mode.

– State 2: There should not exist a state where collision occurs, and also sink decides change of superframe thus change of operational modes. There are two possibilities for this property, for some configurations this is a faulty state, and for others it is not a faulty state. Configurations for which this is a faulty state is discussed later along with topology discussion.

– State 3: Property similar to state 2 but to check for existence of a state where collision and no change of superframe. But, unlike state 2 this property should hold across all configurations.

– State 4: Critical change of state, i.e., from steady to transient is possible in the protocol design. In case of collisions, the nodes save the alert and send an alert again in the next alert slot. Thus, we check also the possibility where the saved alert eventually results in the change of operational modes (steady to transient).

– State 5: Given the dual mode operation, the nodes and the sink should always work on the same mode of operation.

– State 6: In a case where the process stays in the transient state permanently, the protocol needs to stay in transient mode of operation to suit the application needs. Similarly, when a process starts out in the steady state, and remains in the same for the entire duration of process operation, the protocol has to imitate the same via its superframes.

– State 7: A system should be able to reach either of the operational modes whenever required (reachability property).

Before the verification of the listed properties, we have to first validate the verification model against the design. We use Message Sequence Charts (MSC) to validate the presence of the important features of the protocol in the Uppaal model. The features and possibilities include data transmission between nodes, between nodes and sink, carrier sense, and collision of alert messages. Apart from representing these properties via MSC we also validate the protocol model via verification using queries, and is discussed in Sec. 5. Some of the representative properties used as queries to validate the model are:

1. Data communication occurs between sensors, and also reaches the sink.

2. Existence of collision, given the possibility of collision in alert slots (both in nodes and sink). Alternatively, as a negative query we can check for the absence of collision.

3 Uppaal

Uppaal [8] is a tool-set used for model-checking real-time systems, and is based on timed automata. It is an integrated tool environment that supports modeling, validation, verification, and simulation. An abstract representation of a given real-time system is designed as a network of timed automata, validated, and then verified. Most model-checking tools in general allow for qualitative analysis. Along with qualitative analysis Uppaal has extended model-checking to also allow quantitative analysis, which also gives important insights to the performance of the real-time systems. The query language of Uppaal allows for verification of properties like safety, reachability, liveness, and other time-bound properties. The statistical extension of Uppaal (Uppaal-SMC) allows for quantitative analysis which is used for performance verification in protocols like the LMAC protocol [8]. Uppaal consists of two simulators, symbolic and concrete. The symbolic simulator is used to follow the execution of the model step by step. For certain queries, Uppaal outputs traces which can be viewed in the symbolic simulator, this could

help in pin pointing error locations and events leading to errors/faults. The symbolic simulator also shows all the templates in the model, a message sequence chart (MSC) which can be used to represent communications between different processes, and allows interactive step wise simulation of the model. Along with features similar to symbolic simulator, the concrete simulator has added advantages of firing transitions at a specified time. The concrete simulator also produces a gantt chart along with MSC; gantt chart is used to represent schedules in real-time systems.

3.1 Modeling in Uppaal

In this section, we briefly describe the modeling elements and notations used in Uppaal, for better understanding of the model. For extensive detail on modeling in Uppaal refer to [10]. The Uppaal modeling language which is based on timed-automata is an FSM with the concept of time. Clock variables (global and local) are used to represent time. A given real-time system is modeled as a network of such timed automatons. Each independent timed automaton is defined as a template. The system in general can have multiple instantiations of such a template. For example a node model can be a template, and we can have multiple nodes. The Uppaal modeling language consists of three types of declarations: *Global*, *Local/Template*, and *System*. *Global* declarations similar to other plethora of programming languages are declarations that are visible to all the templates within the system. *Local/Template* declarations are visible only to the functions and the automaton within the template. *System* declarations are used to define the template configuration that makes a system, this includes instantiating each template (Multiple nodes and one sink etc.).

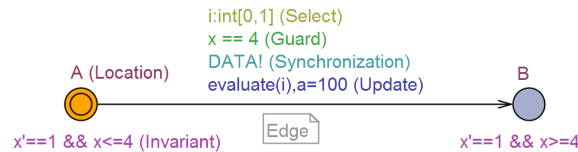

Fig. 6: An example template

A template automaton is made up of a set of *locations*, *edges* between these locations that define transition conditions, and local declarations. An example template is shown in Fig. 6. The example consists of two locations `A` and `B`. The locations are similar to states in FSM, and all template automatons have initial locations marked with an extra inner circle (location `A`), when the execution begins the template starts in the initial location. The locations can have invariants that apply as a condition for the model to be in the particular location. The invariant in location `A` is $x' == 1 \&\& x <= 4$, here x is a clock variable. In Uppaal, clocks progress synchronously, thus clock variables are used to keep template instantiations synchronized. The condition $x' == 1$ defines that the clock is functioning, $x' == 0$ defines pausing the clock. The condition $x <= 4$ defines that clock can stay in the location until it is 4 time units. The edge towards the location `B` consists of four parts: *select*, *guard*, *synchronization*, and *update*. The *select* part consists of one or more select statements of the form $i : int[0, 1]$, the model can then select a value for variable "i" randomly within the specified range. The *guard* part consists of one or more Boolean conditions that have to be true (collectively) in order to be able to take the edge. The *guard* in the example is $x == 4$ which defines that the clock variable has to have a value 4 time units for the guard to be true and the edge to be active. The *synchronization* part consists of channel synchronization variable with a sender and a receiver. In the example, we have a channel synchronization variable *DATA*, and it represents send with the "!" symbol after the variable. Another template automaton that has an edge with the synchronization *DATA?* where "?" defines receive will

synchronize over this channel given that particular edge is active and can be taken. An example receive synchronization is shown in Fig. 7. The second template example consists of two locations `C` and `D` with invariant $y' == 1$ (y is a clock variable). When the edge between the two locations in the first template is taken the edge in the second template is taken due to synchronization channel variable. Lastly, the *update* part includes a list of variables and functions that are updated/executed when the edge is taken. In the example shown in Fig. 6, the function EVALUATE(I), and the assignment statement $a = 100$ are update statements.

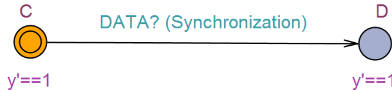

Fig. 7: An example template with channel receiver

Query language Uppaal uses a subset of Timed Computation Tree Logic (TCTL) [10] as its query language to represent properties of the model as queries. Similar to TCTL the query language has path formulae, and state formulae. A state formula is used to represent individual states. An example state formula is $x == 4$, which evaluates to true if the model has a state with $x == 4$. A path formula is used to represent execution paths through the model. Path formulae is used to represent *safety*, *liveness*, and *reachability* properties. A path formula quantifies a state formula over a single path or on all paths. The formula with all paths performs an exhaustive verification throughout the model. Uppaal has two path quantifiers, “E” and “A”. “E” represents an existential quantifier, that searches for one particular path in the system where the property holds, or sometimes just one state where the property holds. “A” represents a universal quantifier, which searches over all paths possible in the system. These quantifiers are used in combination with modalities “<>” and “[]”. “<>” represents a existence (once at least) of a state in the system. “[]” represents invariance over the path quantifier combined with. The combination of these modalities with the path quantifier is used represent various properties as listed below:

– A<> ϕ for all paths ϕ eventually holds or evaluates to true. ϕ here is a state formula.
– A[] ϕ for all paths ϕ always holds or invariantly holds.
– E<> ϕ exists a path where ϕ holds at least once.
– E[] ϕ exists a path where ϕ always holds.

4 DMAMAC Uppaal model

We use a non-deterministic timed model to verify the properties of the DMAMAC protocol on a time based system setting. Certain non-determinism exists in terms of choosing the alert delay for alert messages in nodes, and for the sink to make the decision to change from transient to steady; otherwise rest of the model contains deterministic choices. Given the design of alert messages, collision is possible when sending alert messages. We use a simplified collision model, detailed later in this section. The sink and sensor/actuator nodes have separate models. The model also includes a global variable time to keep track of real-time (Local clocks are reset with each superframe). Below, we list out the assumptions made supporting the design of the Uppaal model. Further, we discuss both the node and the sink model in detail.

4.1 Assumptions

The abstract model represents partly or wholly the behavior of a given system. The main aim of the verification of the protocol is to check working of two modes of operations correctly, along with different possibilities (like collision) that can exist. Various assumptions are made to specify the differences between the original design and the abstract model used for verification. Also, we reason the applicability of these abstractions. The assumptions and design considerations that are made during the design of the Uppaal model for the DMAMAC protocol are:

- Given the focus of the verification on state-switch and related activities, we do not model packet exchange between the nodes explicitly, an abstract representation is used. The messages or packets exchange mechanism is represented by channel synchronization in the Uppaal model.
- The DMAMAC protocol is a TDMA protocol, and requires time-synchronization for proper functioning. The time-synchronization used in WSN is not part of the protocol description. We use the time synchronization mechanism provided in Uppaal via its clock variables to substitute the requirement.
- CSMA is an important part of the DMAMAC protocol. An exact model of CSMA results in a rather complex model (a case study in itself). We use a representative of the CSMA procedure, which should not to be confused with the real CSMA procedure. The representation imitates the possibilities thus provides the effects of actual CSMA on the working of the protocol. The possible effects include skipping packet transmission on detection of ongoing transmission, and also collision possibilities.
- 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.
- We consider ideal channel conditions with no packet failure, thus packet failure is not modeled. ACK timeout is not used since we do not consider any packet failure.
- A protocol based assumption, also used in the model: sensor nodes always send data packets, thus data time out is not used.
- A modeling addition: channel synchronization variable *choice* is used to force enabled transmissions. This is a modeling element and is not part of the protocol. Also, only the send part of the synchronization is used, no receivers exist for this synchronization.

4.2 Sink Model

The sink model is shown in Fig. 8. The model is an expansion on the sink FSM with all conditions and effects. The colors in the automaton adhere to the colors of the states in the FSM. Both sink and node automatons begin with an initial location **Start**. The sink initiates network startup procedure of the network using broadcast synchronization channel *startup* over the edge towards the **StateController** location. This symbolizes a startup procedure, also a function INITIALIZE() is used to set proper values to local and global variables. The sink reaches the **StateController** with the startup of the network. The node automatons synchronize with the channel variable *startup*, and reach the **StateController** location. The **StateController** 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 "$x' == 1$" as an invariant on all states. 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 WSN is 10 milliseconds, thus we use the same unit "ms" for time in our model. Given the time unit of "ms", the variable *currentMaxSlots* is

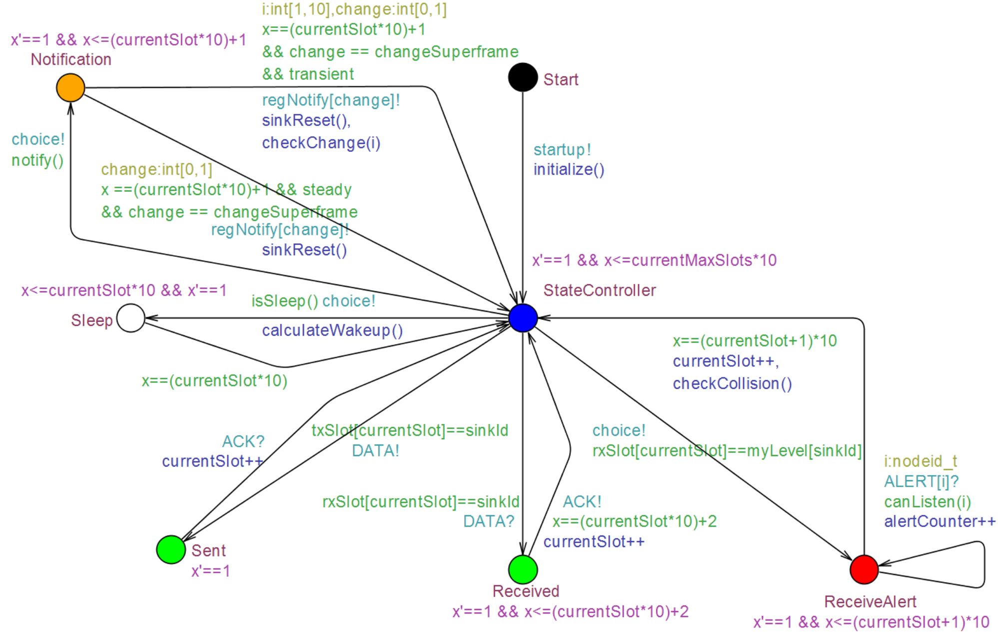

Fig. 8: Uppaal Model of the sink

multiplied by 10 to obtain the corresponding time. The state is also similar to the *self-message* handler in the OMNeT++ [11] 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 model then traverses between the different FSM states or locations in the model 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 based on the current slot in the superframe. The location **Notification** represents the "notification" state in the FSM (steady and transient). The notification message is sent by the sink, and received by other nodes in the network. The notification message sending is mimicked by the broadcast channel *regNotify[change]*, *change* here carries a message of the status of the Boolean variable *changeSuperframe*. The *changeSuperframe* variable is true when the sink needs to indicate a change in superframes to switch the mode of operation to the nodes. For both steady to transient and transient to steady switch, the sink uses *changeSuperframe* to indicate the change. The switch from transient to steady is decided by the sink, in the absence of real inputs we include a random variable to decide if a transient to steady has to happen. Also a select statement, $change : int[0,1]$ select statement is used for modeling purpose (not part of the protocol), and is equated to the *changeSuperframe* in the guard to make sure that the changeSuperframe value is used (not random selections). The other notification edge is used for the steady mode; this is controlled by the guard on these edges. Also, as 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 these edges have a function SINKRESET(), which resets the sink variables at the beginning of a new superframe including changing of superframes ($currentMaxSlots$ reset etc.).

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 urgent broadcast channel *choice* 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. Note that *choice* is only part of the verification model, and not of the protocol design. The location **Sleep** has an invariant $x <= currentSlot * 10$ which indicates that during execution the control can be in the location until 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.

Location **Sent** is reached when the sink sends data in its data slot, it represents the *Sent* state in the sink FSM. The **Received** location is reached when the sink receives any sensor data, and then sends an ACK via channel synchronization. Location **Received** represents the *received* state in the FSM. An example data transfer from a sensor node to the sink in the Uppaal model is shown in Fig. 9. Location **Received** has an invariant $x \leq (currentSlot * 10) + 2$, 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 the delay is applied. On sending or receiving of ACK synchronization, the local variable *currentSlot* is incremented.

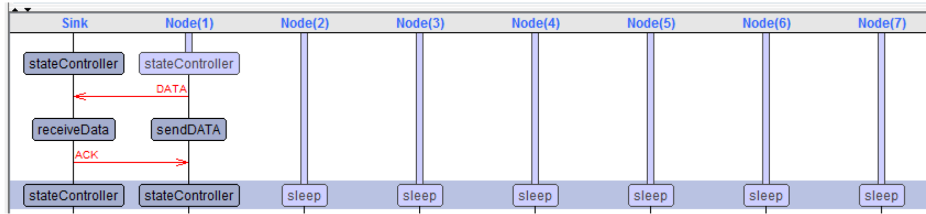

Fig. 9: Message sequence chart for data transfer towards the sink.

Lastly, location **ReceiveAlert** represents "receiveAlert" from the FSM. Location **ReceiveAlert** is reached when it is the sink's turn to receive alert, depending on the alert levels defined in *myLevel* array variable represented by the guard $rxSlot[currentSlot] == myLevel[sinkId]$. The sink stays in the location for 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 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 but the guard makes sure that the sink can listen to that particular node. The listening range represented as a matrix for the nodes is shown below, -1 (indicates none) is used to fill up the matrix otherwise node identification is used (0 is the sink). 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**. An example for the sink receiving an alert message via MSC is

shown in Fig. 10. The collision model used is explained towards the end of this section.

$$ListenMatrix[nodeid][] = \left[\begin{array}{cccc|c} 1 & 2 & 3 & -1 & //Node(0) \\ 4 & 5 & 2 & -1 & //Node(1) \\ 5 & 6 & 1 & 3 & //Node(2) \\ 7 & 6 & 2 & -1 & //Node(3) \\ -1 & -1 & -1 & -1 & //Node(4) \\ 6 & -1 & -1 & -1 & //Node(5) \\ 5 & -1 & -1 & -1 & //Node(6) \\ -1 & -1 & -1 & -1 & //Node(7) \end{array}\right] \quad (1)$$

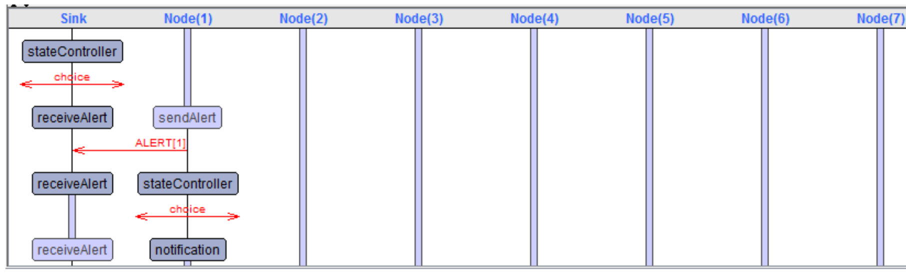

Fig. 10: Message sequence chart for the sink receiving an alert.

4.3 Node Model

The sensor/actuator node model is shown in Fig. 11. The node model is similar to the sink model except for notification handling procedure. Also, the node template consists of an extra location for sending alert messages. The notification part of the node model is simple, since it only receives notification. Location **Notification** is reached when the node is in its notification slot (receive) in either of the superframes. The nodes then synchronize on the channel variable *regNotify[change]* from the sink, and reset the node variables using function NODERESET() based on the value of variable *change*. A message sequence chart for notification sent from the sink, and received by all nodes is shown in figure 12.

The node model works similar to the sink model for sleep, sent (data), received (data), and alert receive. Thus, 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. Location **SendAlert** represents the "sendAlert" state in the FSM (steady). 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 based on the process. We imitate this event using probability weight based selection for sending alert messages as depicted in Fig. 13. The choice of node sending an alert depends on the guards on the edge that makes sure it is the alert slot of the node, and that the node does not already have an alert to be sent. The nodes can already have an alert to be sent in their slot either when they have received an alert from one of their children or when they

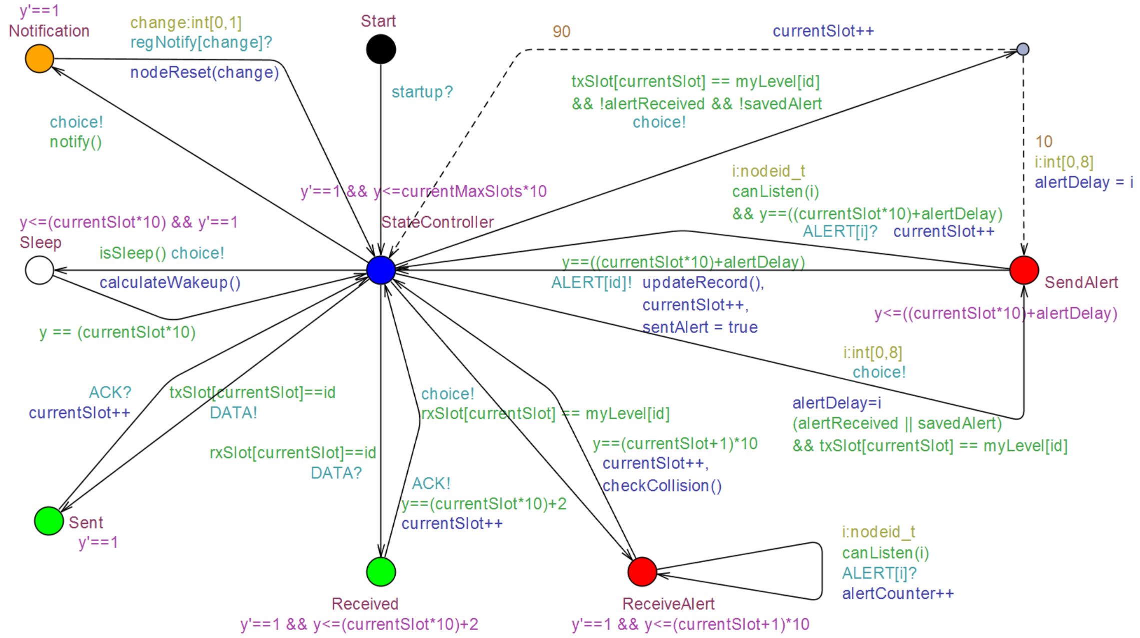

Fig. 11: Uppaal Model of a regular sensor/actuator node

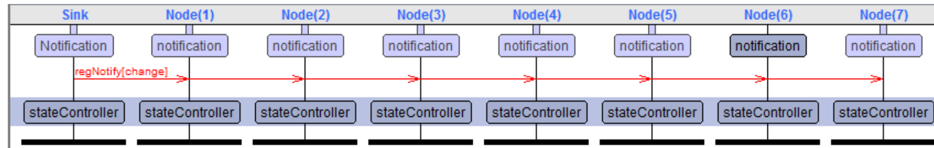

Fig. 12: Message sequence chart for a notification message from the sink

have a saved alert from the previous round. Boolean variables *alertReceived* and *savedAlert* represent these conditions. 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 the location **SendAlert** for the duration of the alert delay, performs carrier sense (representative) prior to sending the alert message. This is represented by the edge with a guarding 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 basically use a representative carrier sense mechanism in the model. We see that the 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 instance 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 when two nodes can hear each other and have the same alert delay, one of the two nodes skips sending the alert message. But, when the nodes do not hear each other and that the receiver can hear both, the packets collide at the receiver. An example message sequence of carrier sense (representative) is shown in Fig. 15. In this example Node(5) and Node(6) are trying to send alert with the same alert delay (3ms) as

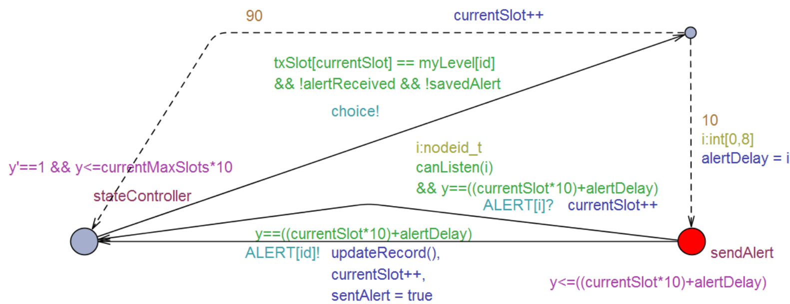

Fig. 13: A part of node model with alert probability for sending alert

shown in List. 1.1. In the listing apart from the trace we have added comments with prefix "//" to add more detail. When Node(5) begins to send the alert, Node(6) senses the same and skips sending alert via the edge guarded by canListen(i) function.

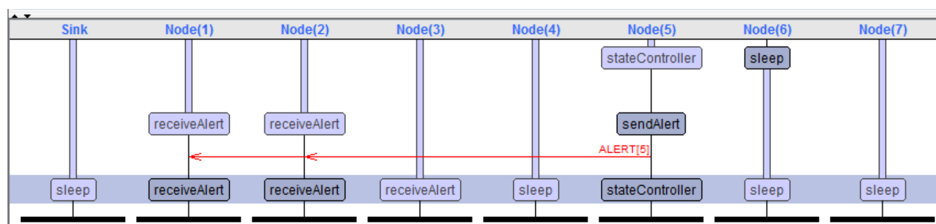

Fig. 14: Message sequence chart for an alert between nodes

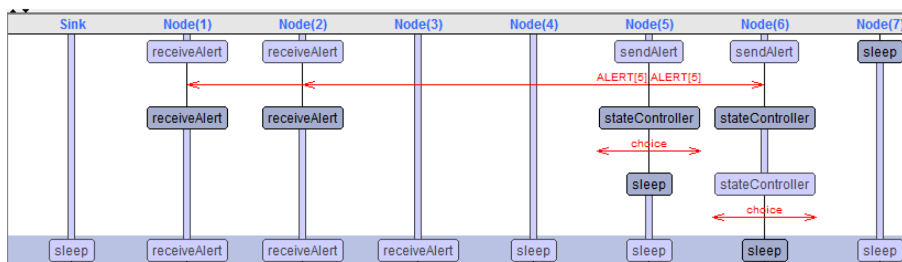

Fig. 15: Message sequence chart for carrier sense during Alert

Listing 1.1: 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)
```

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 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*. 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. An example of alert notification between two nodes is shown in the MSC in Fig. 14. It shows Node(5) sending an alert to Node(2). This message is overheard by Node(1) due to its listening range.

In certain cases when the alert messages fail to reach the sink due to collision, *savedAlert* variable is used to save the alert and are sent in the next round. During this, the probability edge is not used, the nodes directly move to the location **SendAlert** via the edge with the guard ($alertReceived || savedAlert$) as shown in Fig. 16. The variable *alertReceived* represents the case when nodes have to forward 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.

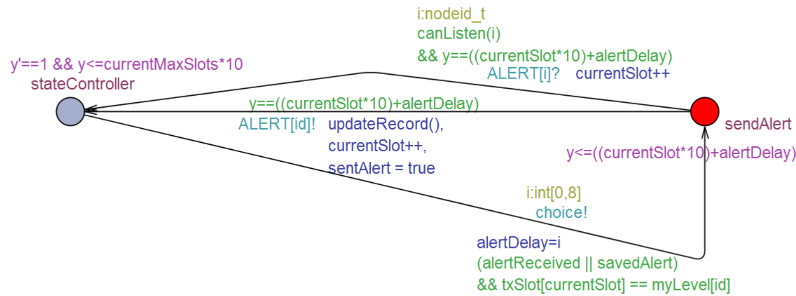

Fig. 16: Second condition for sending alert

4.4 Collision

For the collision model, similar to [7] and [6], we use a simple collision model. In the LMAC [6] protocol, when two nodes send a packet in the same slot it is considered as 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 children, the parents can also listen to nodes in the vicinity. 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. 17. The trace for the same is shown in List. 1.2. For the given case Node(1) and Node(3) choose the same alert delay of 7 (ms) independently, and 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)identify this and save the alert. The saved alert is used to repeat sending alert in the next round (with a new delay) to make sure the superframe changes. Note that there could be a possibility where collision occurs at lower levels, and even at the sink but still change of superframe occurs because of another alert message reaching the sink. For our model we have created a topology in such a way that both cases exist but on different configurations, these configurations are discussed

in 6. 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 they receive nothing at all. A representative model of collision detection is used to show the effects of collision on change of superframe.

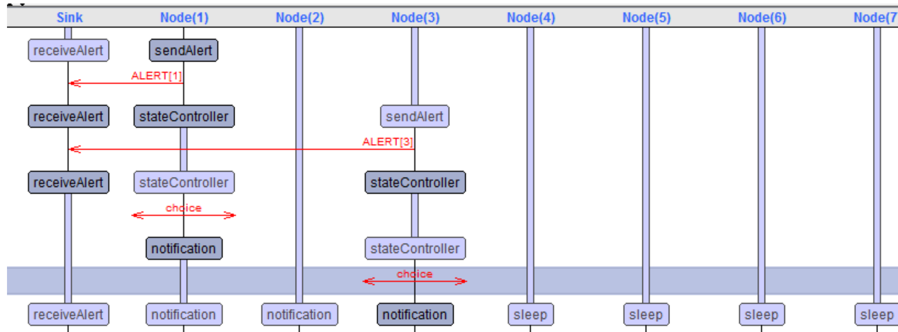

Fig. 17: Message sequence chart for collision at the sink

Listing 1.2: 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)
```

4.5 Node Topology

The node topology used for the verification of the models is shown in Fig. 18. We use 5 sensor nodes, 2 sensor-actuator nodes, and a sink in the given tree topology. We consider a small topology to keep the state-space low to be able to have exhaustive verification possible. The current node topology has 3 ranks but since the sink only listens (and not send alert) we have 2 alert slots based on the DMAMAC protocol. This topology currently allows for both carrier sense (representative) and collision, has both sensors and actuators with data communication for both sides, 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, which 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 than calculating it dynamically based on multiple factors like node position, path-loss, and receiver sensitivity as done in network simulators.

In the current topology, used for the evaluation of the DMAMAC protocol various functionalities that need to be verified are covered. The DMAMAC protocol is used for applications with offline scheduling, with scheduling done pre-deployment all slot allocations are known prior to deployment. The topology in general is well planned, and no random deployment is used. Though, a real topology would be much larger than the one consider here. In the current topology we have 3 levels and maximum of 2 hops.

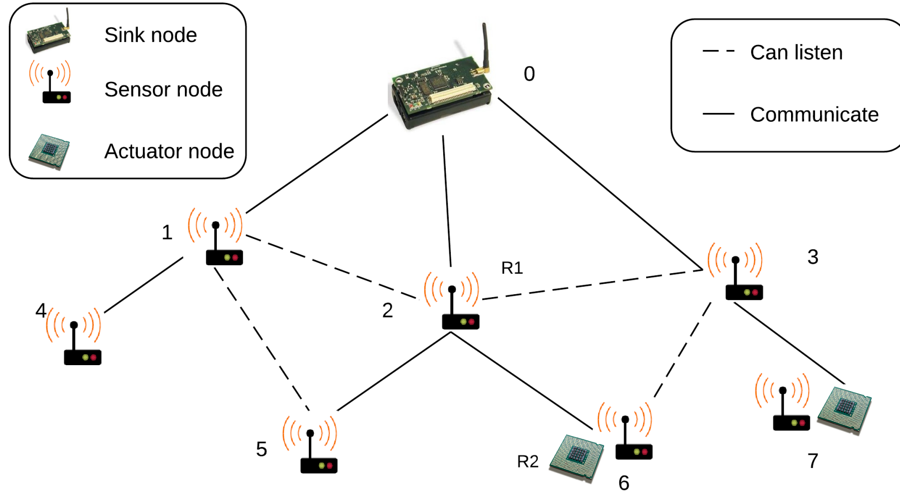

Fig. 18: Node topology

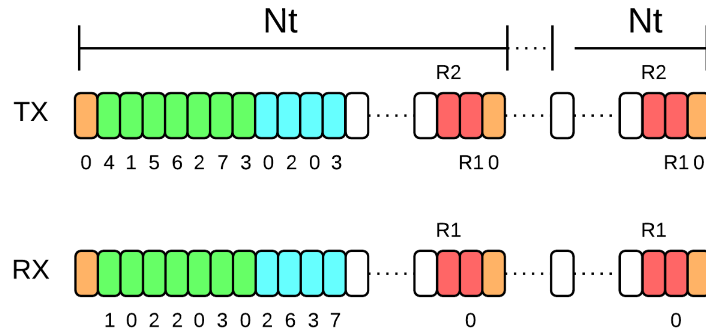

Fig. 19: Superframe structure based on the schedule and node topology

For a qualitative analysis this covers the error scenarios that could potentially exist in multiple-hops. Thus, an n-hop scenario with $n \geq 3$ can be reduced to a 2 hop scenario, and $n \geq 4$ to a 3-hop scenario and so on. This does not result in loss of qualitative analysis applicability. An error that could exist in a $n \geq 3$ scenario would already exist in a 2 hop scenario. Although, the number of child nodes per node stays the same even in larger networks (defined by protocol), the listening range of a given node might be greater than the number considered here (maximum of $n = 4$) depending on the distance between the nodes. But similar to multiple hops situation, a larger listening range $n >= 5$ can be reduced to $n = 4$ from a qualitative analysis perspective. Hence, we argue that the qualitative analysis based on the current topology is applicable to larger networks used by applications.

The schedule for the given node topology is shown in Fig. 19. In the example schedule we skip re-transmission, and the sink processing slots. We assume no packet loss, thus re-transmission slots do not make a difference in the model. 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 message only sender identification (sink) is marked. The schedule only represents the steady superframe, for transient superframe only the first Nt part is used with the alert parts replaced by sleep. Note that, we use 10 ms as slot duration and time is in general taken in “ms” (milliseconds).

4.6 Configurations

Multiple configurations of the DMAMAC protocol are possible based on values that can be varied in the model. Firstly, steady and transient: the Uppaal model could start with either steady or transient and this could have certain effect on the verification, and is discussed in Sec. 6. Another important factor affecting the configurations is the range of possibilities for the variable *alertDelay*. In the protocol we have the range [0,8], the range from which the nodes randomly choose their delay to be applied before sending the alert message. Due to state-space issues we use only $alertDelay[1, 1]$ for exhaustive queries, e.g., deadlock query. This does not mean that $alertDelay[1, 2]$and above have deadlock, but only means that the query results in memory exhausted error. $alertDelay[1, 1]$ in itself covers all possibilities including possibility of state-switch, collision, and CSMA, thus covering the qualitative aspects of the protocol. The increase in $alertDelay[1, 2]$ will not result in deadlock. 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 other configurations is the applicability of State 2 and 3 of the verification properties and is further discussed in Sec. 6. 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, thus does not affect the qualitative results.

5 Validation

Validation is the process of checking that the Uppaal model of the DMAMAC protocol conforms to the protocol specification. Prior to verification of the protocol specification against its design, we need to validate the verification model we built in Uppaal. Important features of the DMAMAC protocol that we used to validate the model via MSC obtained from the symbolic simulator of Uppaal are: data transmission between nodes, also between nodes and the sink, sending/receiving of alert message, possibility of collision, and carrier sense when sending alert messages, was done in previous section 4. Further, we validate using the verification engine of Uppaal by describing the validation properties described in Sec. 2 in the form of Uppaal queries. Below, we detail the properties transformed into queries in the Uppaal query language, and verified them using the symbolic verifier. Also, given that we have two versions of the model depending on which operating mode (steady or transient) the model execution starts with, we verify these properties on both model variations.

- Data transfer (nodes) : A<> exists (i:nodeid_t) exists (j:nodeid_t) (i≠j && Node(i).Send && Node(j).Receive)
  This query represents the validation property 1, that the system can potentially always have data communication between two different nodes. This property is satisfied for both variations of the model.
- Data transfer (towards the sink): A<> exists (i:nodeid_t)(Node(i).Send && Sink.Receive)
  Also represents validation property1, to check if the sensor data is also reaching the sink. This property is satisfied for both variations with different starting mode.
- Collision existence (nodes): E<> exists (i:nodeid_t) Node(i).collision
  This property describes validation property 2 describing that collisions do happen, which is true by design. We can possibly only use potentially exists path query since collisions may happen, and need not happen always like data transfer. Thus, we just check when collisions do occur they are detected by nodes successfully. This is a reachability property, and is satisfied for both variations.
- Collision existence (At the sink): E<> exists Sink.collision

This query represents validation property 2 to check collision existence at the sink. Property is satisfied (for both variations) and an example trace is provided for the same by the tool.

Apart from using MSC and validation queries, we have used the simulator to check for smooth working of the protocol step by step. Details on the performance of the queries on different configurations of the model, given the model-checking tool and hardware used are shown in Appendix A.

6 Verification

The protocol model in Uppaal is verified using the properties defined in Sec. 2. Firstly, we discuss each of the properties with the Uppaal query language representation that applies to both model variations. Later, both model variations with specific properties are discussed separately.

- No deadlock : A[] not deadlock
  The query represents the classic deadlock check, a safety property. This is an exhaustive query, thus is checked only for the configuration with $alertDelay[1, 1]$. This query applies for both variations of the model and is satisfied.
- State 1a (faulty): E<> not (Sink.steady || Sink.transient)
  This represents a state where the system is not in either of the operating modes. This query when satisified will generate an example trace providing a path where this is possible. For both variations of our model this property is falsified, re-iterating the fact the protocol is always in one of the two possible states and hence is safe.
- State 1b : A[] forall $(i : nodeid_t)$ (Node(i).transient == Sink.transient || Node(i).steady == Sink.steady)
  An exhaustive query to check if all nodes and sink have the same operation mode at every instant during execution. The query results to true, thus the property is satisfied.
- State 2 : E<> (Sink.collision && Sink.changeSuperframe)
  We represent the property of the protocol to see if the changeSuperframe is still set to true if the sink detects a collision. In reality we could have a system with the DMAMAC protocol applied, where collision occurs but still a change of superframe (mode) occurs. This is possible in a case for example, node 1 and node 3 send alerts but collide due to same alert delay, at a later time node 2 sends an alert which is successful. But in the configuration with $alertDelay[1, 1]$ where all nodes can only pick one possible alert delay (1 ms) if sending any alert, this property would evaluate to false (or is a faulty state). For configurations with $alertDelay[1, 2]$ and above the property evaluates to true since there exists states where the sink observes collision but also changes superframe because of an alert received later or before collision. Thus, the result of this query changes with the configuration, and evaluates to expected result.
- State 3 : E<> (Sink.collision && !Sink.changeSuperframe)
  Property represents a state where a collision prevents alerts from inducing change of superframe (representing the case where alert was not received due to collision). This property is satisfied across all configurations.
- State 4 (Critical change of state) : E<> exists (i:nodeid_t) Node(i).savedAlert && Sink.steady && Sink.changeSuperframe
  This query represents a property of the protocol where if an alert fails to induce change of superframe due to collision, the alert is then saved (savedAlert). The saved alert is used again eventually resulting in a change in superframe to transient (from steady). We use the query which searches for one example where this occurs since, given the nature of the model, the collisions could

occur forever preventing the change of superframe in principle. The property is satisfied and an example simulation trace for the possibility is obtained.

An example trace for the collision and change of superframe to co-exist is shown in listing 1.3. In this example the node 1 and node 3 choose the same alert delay (1 ms), also they cannot listen to each other thus 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`. The query representing State 1a (faulty) 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 $i : int[1,2]$. We obtained expected results for all these configurations.

Listing 1.3: 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)
```

6.1 Transient version

Here we list the properties that apply only to the model version starting with the transient mode.

- State 5: E[] transient : The given query checks if there is a path where the system can invariantly be in transient mode. This property is satisfied. For the model with starting mode as steady mode, it is false by default on the starting state.
- State 6: E<> steady : Represents the reachability of steady mode from the starting state. The property evaluates to true.

6.2 Steady version

Here we list the properties that apply only to the version starting with the steady mode.

- State 5: E[] steady : Similar to the query in transient version this query represents a path where steady mode is invariantly true. This property is satisfied.
- State 6: E<> transient : Represents the reachability of transient mode from the starting state (started with steady mode). The property evaluates to true. This property is particularly important since the system should be able to reach transient mode when starting from the steady mode, since in the case where the system starts with transient mode this property is satisfied by default.

Details on the time used by the queries to execute based on different configurations of the model, given the model-checking tool, and hardware used are shown in Appendix A.

7 Conclusion and Future Work

In this report, we have detailed the verification process of the DMAMAC protocol, a protocol designed for process control applications. We used the model-checking tool Uppaal for verification. We have created a network of timed automata with multiple nodes and a sink working on the DMAMAC protocol. We have explained the model in Uppaal including its modeling elements and templates in detail. Also, we have validated the model against the protocol specification. Validation was done using message sequence chart representation of important features, and possibilities of the protocol including data transfer, alert message functioning, carrier sense, and possibility of collision. Further, we also used these features as queries to validate the model. The validated model was then verified for the switch procedure, safety properties including absence of deadlock and other faulty states. Two versions 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 to verify the queries. Performance of these queries on different configurations based on the model created, tool and hardware used is detailed in the appendix to give a better understanding of the possibilities. We used a representative node topology that covers all important features of the protocol including existence of sensors and actuators, multihops, alert messages, and possibility of collision. The DMAMAC protocol model in Uppaal tested positive to the verification done thus increasing the confidence on the design of the protocol. We have argued that the results would remain unchanged in case the node topology is changed, or higher number of sensor/actuator nodes are used given that we cover the main features of the protocol in the topology presented. As a proposed future work, a stochastic model of the DMAMAC protocol to verify the quantitative properties including collision probability and energy consumption could provide further insights to the working of the protocol.

References

1. Akyildiz, I.F., Kasimoglu, I.H.: Wireless Sensor and Actor Networks: Research challenges. Ad Hoc Networks **2** (2004) 351–367
2. Hespanha, J.P., Naghshtabrizi, P., Xu, Y.: A survey of recent results in Networked Control Systems. Volume 95. (2007) 138–162
3. Kumar S., A.A., Øvsthus, K., M. Kristensen, L.: 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. (2014)
4. Billington, J., Gallasch, G., Han, B.: A Coloured Petri Net approach to protocol verification. In: Lectures on Concurrency and Petri Nets. Volume 3098 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2004) 210–290
5. Fehnker, A., van Glabbeek, R., Hfner, P., McIver, A., Portmann, M., Tan, W.: Automated analysis of AODV using Uppaal. In Flanagan, C., Knig, B., eds.: Tools and Algorithms for the Construction and Analysis of Systems. Volume 7214 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 173–187
6. Fehnker, A., van Hoesel, L., Mader, A.: Modelling and Verification of the LMAC protocol for Wireless Ssensor Networks. In Davies, J., Gibbons, J., eds.: Integrated Formal Methods. Volume 4591 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2007) 253–272
7. Tschirner, S., Xuedong, L., Yi, W.: 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, ACM (2008) 69–78
8. David, A., Larsen, K.G., Legay, A., Mikucionis, M., Poulsen, D.B., Vliet, J., Wang, Z.: Statistical Model Checking for networks of Priced Timed Automata. In: Proceedings of the 9th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS). Volume 6919 of LNCS., Springer Berlin Heidelberg (2011) 80–96

9. Suriyachai, P., Brown, J., Roedig, U.: Time-critical data delivery in Wireless Sensor Networks. In: Proceedings of DCOSS. (2010) 216–229
10. Behrmann, G., David, A., Larsen, K.: A tutorial on Uppaal. In Bernardo, M., Corradini, F., eds.: Formal Methods for the Design of Real-Time Systems. Volume 3185 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2004) 200–236
11. Varga, A., Hornig, R.: An overview of the OMNeT++ simulation environment. In: SIMUTOOLS. (2008) 60:1–60:10

A Performance

Given the complexity of the model and the limitations of the tools in terms of exhaustive verification of models we present certain performance measures for different configurations that were used during verification. The obtained performance is mainly due to the complexity introduced in the model of the protocol and the hardware limitations in this case (4 GB RAM used). The version of Uppaal used was v4.1.19 (latest release). Firstly, we list performance measures for validation queries in Tab. 1 and Tab. 2. Certain properties have similar performance over both variations of the model, and are mentioned specifically in the transient variation table.

Query	Result	CPU Time (s)	Memory Resident/Virtual(KB)
alertDelay[1,1], i:int[1,2]			
Data transfer (nodes)	Satisfied	0.078	62,340/137,100
Data transfer (sink)	Satisfied	0.031	14,652/58,524
Collision existence (nodes)	Satisfied	1.092	16,008/43,644
Collision existence (sink)	Satisfied	0.561	17,428/45,064
alertDelay[1,2], i:int[1,2]			
Data transfer (nodes)	Satisfied	0.047	12,992/55,292
Data transfer (sink)	Satisfied	0.062	13,080/55,420
Collision existence (nodes)	Satisfied	2.528	18,028/45,724
Collision existence (sink)	Satisfied	2.262	36,596/82,700
alertDelay[1,4], i:int[1,2]			
Data transfer (nodes)	Satisfied	0.063	13,064/55,504
Data transfer (sink)	Satisfied	0.062	15,300/60,828
Collision existence (nodes)	Satisfied	9.782	35,168/68,960
Collision existence (sink)	Satisfied	36.92	499,188/1,004,356

Table 1: Performance of the steady variation of the model on the validation queries

The verification queries on both variations of the model, and model variation specific queries are listed in Tab. 3 and Tab. 4. We mainly varied the variable *alertDelay* between [1,4], in the protocol this can be extended to [0,8] of a given slot. The select statement $i : int[0, 8]$ on the notification edge of the transient mode was reduced to $i : int[1, 2]$ for easier verification. The only difference this makes from the real protocol is that now the probability of choosing steady mode after the transient mode

Query	Result	CPU Time (s)	Memory Resident/Virtual(KB)
alertDelay[1,1], i:int[1,2]			
Similar to steady with same configuration			
alertDelay[1,2], i:int[1,2]			
Data transfer query results are similar to steady with same configuration			
Collision existence (nodes)	Satisfied	6.505	24,528/55,196
Collision existence (sink)	Satisfied	5.96	61,840/129,452
alertDelay[1,4], i:int[1,2]			
Data transfer query results are similar to steady with same configuration			
Collision existence (nodes)	Satisfied	47.799	102,224/206,340
Collision existence (sink)	Satisfied	79.966	986,448/1,971,616

Table 2: Performance of the transient variation of the model on the validation queries

is 50% instead of the planned 90% by protocol design. In reality, this completely depends on the input from the process. This does not have an effect on the verification results of the protocol. For verification purpose we believe it suffices to provide three variants of *alertDelay* which covers the case of multiple delay possibilities. The table is provided to obtain an idea of which properties are verifiable over different configurations based on the current model. Also note that "State 2" which is faulty for the model with $alertDelay[1,1]$ is not a faulty state for other configurations since with multiple alert delays, collision and change of superframe can appear in the same state. Note that the time consumed for verification can vary based on the order in which queries are verified. For two queries that seek similar state-space, the second query execution can take far less time than the first when executed second (continuously), but when executed independently the second query could take up execution time equal to the first query. Ex. State 2 and State 3 when executed continuously take 752 s and 258 s respectively. On independent execution (restarted tool) State 2 takes 752 s and State 3 takes 762 s.

Query	Result	CPU Time (s)	Memory Resident/Virtual(KB)
alertDelay[1,1], i:[1,1], without guard $y == ((currentSlot * 10) + alertDelay)$			
No deadlock	Satisfied	754.312	1,130,624/2,267,656
alertDelay[1,1], i:[1,2], without guard $y == ((currentSlot * 10) + alertDelay)$			
No deadlock	Satisfied	1,338.176	1,906,604/3,848,228
alertDelay[1,1], i:[1,2], Generic queries			
State 1a (faulty)	Not Satisfied	752.003	1,928,672/3,870,476
State 1b	Satisfied	822.126	1,928,788/3,870,576
State 2 (faulty)	Not Satisfied	762.205	1,809,512/3,636,084
State 3	Satisfied	1.261	16,272/42,764
State 4 (Critical change of state)	Satisfied	16.49	62,336/137,092
alertDelay[1,1], i:[1,2], Steady specific queries			
E[] Sink.steady	Satisfied	0.032	17,424/58,892
E<> Sink.transient	Satisfied	1.965	19,308/48,796
A<> Sink.steady	Satisfied	0.016	19,308/48,796
alertDelay[1,2], i:[1,2], Generic queries			
State 1a (faulty)	N/A	N/A	Memory exhausted
State 2 (not faulty here)	Satisfied	3.791	35,836/79,948
State 3	Satisfied	3.9	36,464/82,492
State 4 (Critical change of state)	N/A	N/A	Memory exhausted
alertDelay[1,1], i:[1,2], Steady specific queries			
E[] Sink.steady	Satisfied	0.046	36,596/82,700
E<> Sink.transient	Satisfied	16.708	66,116/137,096
A<> Sink.steady	Satisfied	0.109	66,116/137,096
alertDelay[1,4], i:[1,2], Generic queries			
State 2 (not faulty here)	Satisfied	43.836	557,076/1,116,984
State 3	Satisfied	43.961	498,860/1,003,768
alertDelay[1,4], i:[1,2], Steady specific queries			
E[] Sink.steady	Satisfied	0.032	13,916/55,452
E<> Sink.transient	Satisfied	453.573	1,576,432/3,149,952
A<> Sink.steady	Satisfied	0.015	13,940/40,820

Table 3: Performance of the steady variation of the model on the tool based on queries

Query	Result	CPU Time (s)	Memory Resident/Virtual(KB)
alertDelay[1,1], i:[1,1], without guard $y == ((currentSlot * 10) + alertDelay)$			
No deadlock	Satisfied	688.042	956,108/1,911,188
alertDelay[1,1], i:[1,2], without guard $y == ((currentSlot * 10) + alertDelay)$			
No deadlock	Satisfied	1,133.925	1,789,908/3,589,064
alertDelay[1,1], i:[1,2], Generic queries			
State 1a (faulty)	Not Satisfied	718.837	1,795,596/3,595,148
State 1b	Satisfied	876.586	1,772,436/3,573,820
State 2 (faulty)	Not Satisfied	711.287	1,797,488/3,599,136
State 3	Satisfied	3.214	21,044/49,512
State 4 (Critical change of state)	Satisfied	33.868	113,084/238,116
alertDelay[1,1], i:[1,2], Transient specific queries			
E[] Sink.transient	Satisfied	0.015	13,820/56,924
E<> Sink.steady	Satisfied	0.64	13,888/40,168
A<> Sink.transient	Satisfied	0.016	13,888/40,168
alertDelay[1,2], i:[1,2], Generic queries			
State 1a (faulty)	N/A	N/A	Memory exhausted
State 2 (not faulty here)	Satisfied	8.331	62,292/128,008
State 3	Satisfied	8.346	61,616/129,064
State 4 (Critical change of state)	N/A	N/A	Memory exhausted
alertDelay[1,2], i:[1,2], Transient specific queries			
Results are similar to the previous configuration			
alertDelay[1,4], i:[1,2], Generic queries			
State 2 (not faulty here)	Satisfied	85.099	1,100,824/2,210,440
State 3	Satisfied	77.127	986,476/1,971,668
alertDelay[1,4], i:[1,2], Transient specific queries			
Results are similar to the previous configuration			

Table 4: Performance of the transient variation of the model on the tool based on queries