Adaptive Formal Framework for WMN Routing Protocols

Mojgan Kamali[1(✉)] and Ansgar Fehnker[2(✉)]

[1] Åbo Akademi University, Turku, Finland
[2] University of Twente, Enschede, The Netherlands
mojgan.kamali@abo.fi, ansgar.fehnker@utwente.nl

Abstract. Wireless Mesh Networks (WMNs) are self-organising and self-healing wireless networks that provide support for broadband communication without requiring fixed infrastructure. A determining factor for the performance and reliability of such networks is the routing protocols applied in these networks. Formal modelling and verification of routing protocols are challenging tasks, often skipped by protocol designers. Despite some commonality between different models of routing protocols that have been published, these models are often tailored to a specific protocol which precludes easily comparing models. This paper presents an adaptive, generic and reusable framework as well as crucial generic properties w.r.t. system requirements, to model and verify WMN routing protocols. In this way, protocol designers can adapt the generic models based on protocol specifications and verify routing protocols prior to implementation. This model uses Uppaal SMC to identify the main common components of routing protocols, capturing timing aspect of protocols, communication between nodes, probabilities of message loss and link breakage, etc.

1 Introduction

Wireless Mesh Networks (WMNs) are self-organising and self-healing wireless networks that provide support for broadband communication without requiring fixed infrastructure. They provide rapid and low-cost network deployment and have been applied in a wide range of application areas such as public safety, emergency response networks, battlefield areas, etc.

A determining factor for the performance and reliability of such networks is the routing protocols applied in these networks. Routing protocols specify the way of communication among nodes of the network and find appropriate paths on which data packets are sent. They are grouped into two main categories: proactive and reactive routing protocols. Proactive protocols rely on the periodic broadcasting of control messages through the network (time-dependent) and have the information available for routing data packets. Reactive protocols, in contrast, behave on-demand, meaning that when a packet targeting some destination is injected into the network they start the route discovery process.

© Springer Nature Switzerland AG 2018
K. Bae and P. C. Ölveczky (Eds.): FACS 2018, LNCS 11222, pp. 175–195, 2018.
https://doi.org/10.1007/978-3-030-02146-7_9

Previous studies of routing protocols mostly rely on simulation approaches and testbed experiments. These are appropriate techniques for performance analysis but are limited in the sense that it is not possible to simulate systems for all possible scenarios. Formal techniques, mathematically languages and approaches, are used to complement testbed experiments and simulation approaches. They provide tools to design and verify WMN routing protocols, allow to model protocols precisely and to provide counterexamples to diagnose their flaws.

Statistical Model checking (SMC) combines model checking with simulation techniques to overcome the barrier of analysing large systems as well as providing both qualitative and quantitative analysis. Uppaal SMC monitors simulation traces of the system and uses sequential hypothesis testing or Monte Carlo simulation (for qualitative and quantitative analysis respectively) to decide if the intended system property is satisfied with a given degree of confidence. Statistical model checking does not guarantee a 100% correct result, but it is able to provide limits on the probability that an error occurs. In this work, we apply Uppaal SMC [7], the statistical extension of Uppaal.

Contributions. Our work has been inspired by the fact that formal modelling and verification of routing protocols seem challenging tasks. Protocol designers often decide on skipping this level of development. We provide an adaptive, generic and reusable framework as well as crucial generic properties w.r.t. system requirements, to model and verify WMN routing protocols. In this way, protocol designers can adapt the generic models based on protocol specifications and verify routing protocols prior to implementation.

In particular, this study describes how to build reusable components within the constraints imposed by the Uppaal modelling language. It identifies the main components that routing protocols have in common, and how to map them to data structures, processes, channels, and timed automata in the Uppaal language. We show the validity and applicability of our models by modelling Better Approach To Mobile Ad-hoc Networking (BATMAN) [19], Optimised Link State Routing (OLSR) [5], and Ad-hoc On-demand Distance Vector version2 (AODVv2) [21] protocols using our framework.

Outline: The paper is structured as follows: in Sect. 2, we give an overview of the formal modelling language used in this paper. Then in Sect. 3, we shortly overview the general structure of WMN routing protocols. Section 4 is the core of this paper where we discuss our generic Uppaal framework as well as our generic Uppaal properties. Section 5 demonstrates the adaptability of our framework, sketching examples of BATMAN, OLSR, and AODVv2 protocols. We discuss related work in Sect. 6 and draw conclusions as well as propose future research directions in Sect. 7.

2 Modelling Language

Most routing protocols of WMNs have complex behaviour, e.g., real-time behaviour, and formal modelling and verification of such systems end up being

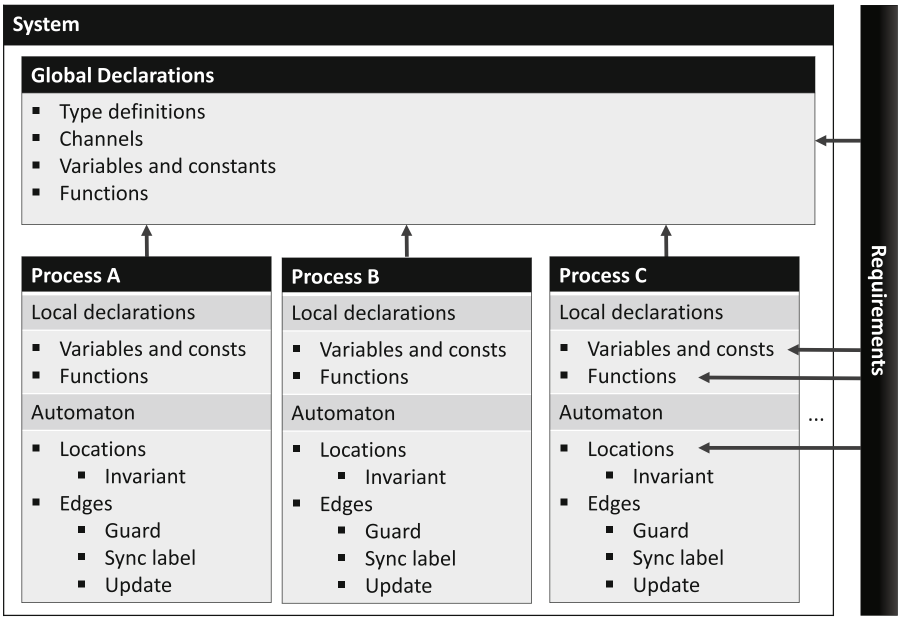


Fig. 1. Common structure of Uppaal models. Arrow denote access to variables and functions. Location names are treated like Booleans by the requirements.

a challenging task. Moreover, choosing a formal modelling language which considers the significant characteristic of these protocols is an important step in the development of a formal framework. In this work, we apply Uppaal SMC (which is based on stochastic time automata) to be able to realistically model timing behaviour, wireless communication, probabilistic behaviour, and complex data structure of routing protocols. In addition, the Uppaal GUI and Uppaal simulator provide a visualised interpretation of the system which makes the task of modelling easier. We describe the basic definitions that are used in Uppaal SMC.

2.1 Uppaal Timed Automata

The theory of timed automata [1] is applied for modelling, analysing and verifying the behaviour of real-time systems. A finite timed automaton is defined as a graph consisting of finite sets of locations and edges (transitions), together with a finite set of clocks having real values. The logical clocks of automata are initialised with zero and are increased with the same rate. Each location may have an invariant, and each edge may have guards (possibly clock guards) which allow a transition to be taken, and/or actions that can be updates of some variables and/or clocks.

The modelling language of Uppaal extends timed automata as defined by Alur and Dill [1] with various features, such as types, data structures, etc [2]. A system is a network of timed automata that can synchronise on channels and shared variables. Fig. 1 depicts the common structure of Uppaal models. Uppaal distinguishes between global declarations and processes. All processes are running concurrently at the same level, and the model has no further hierarchy. Processes are actually instantiations of parameterised templates. Separate from the system model are the requirements, which describe properties, or statistical experiments.

Type definitions in the global declaration are used to define ranges of integers – often used as identifiers – or structs. Variables can be any integer type, any newly defined type, channels, and arrays of these. Clock variables that evaluate to real numbers are used to measure time. All clocks progress at the same rate, and can only be reset to zero. The global declaration can also define functions in a C-like language. These functions can be used anywhere in the model.

Each process has its own local declarations. These may contain variable declarations and function declarations. The scope of these is limited to the process. While it is possible to locally define types and channels, this is rarely done; at most to define urgent broadcast channels that force transitions once their guard becomes true.

For each process, there exists an automaton that operates on local and global variables and may use the locally and globally defined functions. Every automaton can be presented as a graph with locations and edges. Each location may have an invariant, and each edge may have a guard, a synchronisation label, and/or an update of some variables.

Synchronisation between automata happens via channels. For every channel a there is one label $a!$ to identify the sender, and $a?$ to identify receivers. Transitions without a label are internal; all other transitions use either binary handshake or broadcast synchronisation. Uppaal SMC supports only broadcast channels [7]:

Broadcast synchronisation means that one automaton with a !-edge synchronises with several other automata that all have an edge with a relevant ?-label. The initiating automaton is able to change its location, and apply its update if and only if the guard on its edge is satisfied. It does not need a second automaton to synchronise with. Matching ?-edge automata must synchronise if their guards evaluate to true in the current state. They will change their location and update their states. First, the automaton with the !-edge updates its state, then the other automata follow. If more than one automaton can initiate a transition on a !-edge, the choice is made non-deterministically.

Due to the structure of the Uppaal model, automata cannot exchange data directly. A common workaround is the following: If an automaton wants to send data to another automaton, it synchronises on a channel. It writes the data to a global variable during an update, which is then copied by the second automaton to its local variable during its update.

Also, due to the scoping rules, one automaton cannot use a method of one of the other automata, for example, to query its state. The common workaround is to make either a duplicate of important information global, or to have the information global, and have a self-imposed rule on which a process can read and write and to what part of the global variables.

In addition to the system model, it is possible to define requirements. Requirements can access all global and local variables, and use global and local functions, as long as they are side-effect free, i.e. they do not change variables outside of the scope of the function. Requirements can iterate over finite ranges, using `forall`, `exists`, or `sum` iterators.

Uppaal has several other keywords to define the behaviour of delays and transitions, such as `urgent` or `priority`. The discussion of these is outside of the scope of this paper. The common structure of Uppaal, with its scoping rules, however, is relevant for this paper, as it sets the framework in which we have to develop our generic model.

2.2 Uppaal Stochastic Timed Automata

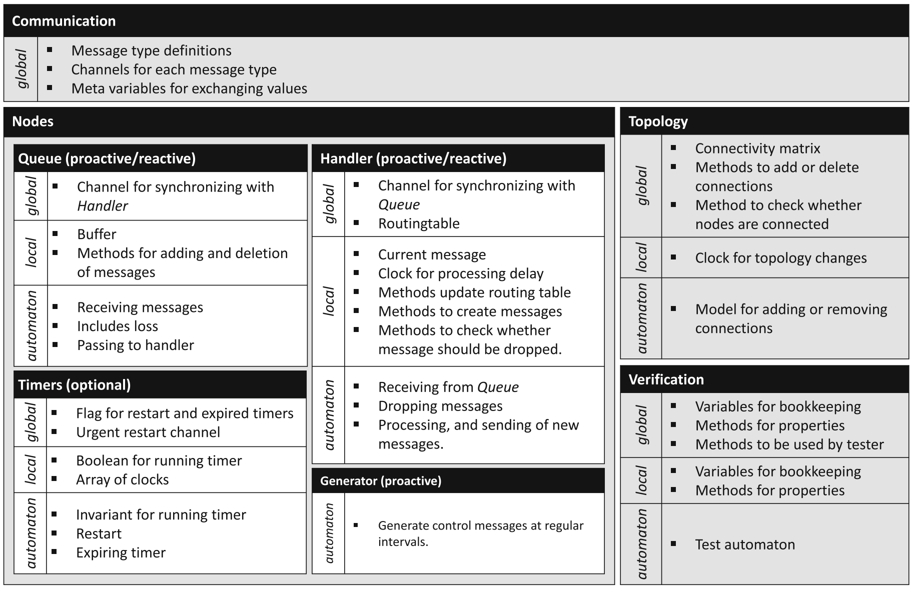
Uppaal SMC [7] is a trade off between classical model checking and simulation, monitoring only some simulation traces of the system and uses sequential hypothesis testing or Monte Carlo simulation (for qualitative and quantitative analysis respectively) to determine whether or not the intended system property is satisfied with a given degree of confidence.

The modelling formalism of Uppaal SMC is based on the extension of Uppaal timed automata described earlier in this section. For each timed automata component, non-deterministic choices between several enabled transitions assigned by probability choices, refine the stochastic interpretation. A model in Uppaal SMC can consist of a network of stochastic timed automata that communicate via broadcast channels and shared variables.

Classical Uppaal's verifier uses a fragment of Computation Tree Logic (CTL) to model system properties. Uppaal SMC adds to its query language elements of the Metric Interval Temporal Logic (MITL) to support probability estimation, hypothesis testing, and probability comparison, and in addition the evaluation of expected values [7].

The algorithm for probability estimation [12] computes the required number of runs to define an approximation interval $[p - \epsilon, p + \epsilon]$ where p is the probability with a confidence 1-α. The values of ϵ (probabilistic *uncertainty*) and α (*false negatives*) are selected by the user and the number of runs is calculated using the Chernoff–Hoeffding bound. In Uppaal SMC, the query has the form: $\texttt{Pr[}\mathit{bound}\texttt{]}(\phi)$, where *bound* shows the time bound of the simulations and ϕ is the expression (path formula).

Evaluation of expected values of a max of an expression which can be evaluated to a clock or an integer value is also supported by Uppaal SMC. In this case, the *bound* and the number of runs (N) are given explicitly and then max of the given expression (*expr*) is evaluated. The query has the form: $\texttt{E[}\mathit{bound};N\texttt{]}(\texttt{max} : \mathit{expr}))$; an explicit confidence interval is also required for these type of queries.



Communication

global
- Message type definitions
- Channels for each message type
- Meta variables for exchanging values

Nodes

Queue (proactive/reactive)

global
- Channel for synchronizing with *Handler*

local
- Buffer
- Methods for adding and deletion of messages

automaton
- Receiving messages
- Includes loss
- Passing to handler

Timers (optional)

global
- Flag for restart and expired timers
- Urgent restart channel

local
- Boolean for running timer
- Array of clocks

automaton
- Invariant for running timer
- Restart
- Expiring timer

Handler (proactive/reactive)

global
- Channel for synchronizing with *Queue*
- Routingtable

local
- Current message
- Clock for processing delay
- Methods update routing table
- Methods to create messages
- Methods to check whether message should be dropped.

automaton
- Receiving from *Queue*
- Dropping messages
- Processing, and sending of new messages.

Generator (proactive)

automaton
- Generate control messages at regular intervals.

Topology

global
- Connectivity matrix
- Methods to add or delete connections
- Method to check whether nodes are connected

local
- Clock for topology changes

automaton
- Model for adding or removing connections

Verification

global
- Variables for bookkeeping
- Methods for properties
- Methods to be used by tester

local
- Variables for bookkeeping
- Methods for properties

automaton
- Test automaton

Fig. 2. Generic structure of a WMN routing protocol model for verification, with respect to typical structure of an Uppaal model.

3 Overview Uppaal Model of WMN Routing Protocols

WMN routing protocols disseminate information in the network to provide the basis for selecting routes. Routing protocols specify which control messages should be sent through the network. These messages are received/lost by other network nodes. Receiving nodes update their information about other nodes based on received messages. Network nodes can send data packets to destinations in the network using discovered paths. Figure 2 depicts the main components that define a routing protocol model.

Communication. The protocol has to specify the types of control messages and the information they contain. This information consists of originator address, originator sequence number, etc. The model has to specify whether a message is sent as a unicast or as a multicast message[1].

Topology. The network topology shows how nodes are connected to each other. Connectivity is commonly modelled as an adjacency matrix. The model should provide methods to add and delete connections, as well as a model that determines how the topology changes. This paper uses a simple model of link failure; a more elaborate study of dynamic topologies can be found in [10].

[1] To avoid confusion, we will refer to this type of communication as *multicast*, instead of *broadcast*. We reserve the term *broadcast* for Uppaal channels.

Node. The behaviour of a protocol is defined by the composition of a queue, a handler, and – if the protocol is proactive – by one or more generators, and possibly a number of timers.

Queue. Messages from other nodes should be stored in a buffer or queue. Based on the order of arrival they will be processed later by the handler (first in, first out method is applied for buffers). If different messages arrive to a node at the same time, the choice of the message reception happens non-deterministically. It will use the information in these messages to update the corresponding routing information about sender nodes.

Handler. The handler is the core of the protocol. The handler will receive messages from the queue and update the stored information, which is kept in the routing table. Depending on the content of the message, and the current state of the routing table, the handler further decides whether to drop the message, send a broadcast message to all other neighbouring nodes, or send a unicast message to another node (both broadcast and unicast message consider sending delays).

Different ways of nodes communication (multicast and unicast) are modelled illustrated schematically in Fig. 3. For multicast synchronisation, we use an array of broadcast channels `Multicast[]`, one channel for each node. The invariant and the guard will encode the timing and duration of a transition (message delay). The guard of the corresponding edge of the receiving node – which will be part of its queue model– will encode the connectivity, i.e., it will not synchronise if the nodes are not connected. The sender will take the edge, regardless of whether other nodes are connected.

Unicast is modelled by a two-dimensional array of channels `Unicast[][]`, one channel for each pair of nodes. Unicast messages assume that on a lower level reception is acknowledged. If this fails, for example, if the nodes are not connected, the sender has to take an alternative transition. A typical alternative would be to multicast an error message or initiate a route request.

Generator. Proactive protocols send control messages at regular intervals. They highly depend on on-time broadcasting of their control messages in order to keep track of network information. Hence, each node includes also a model to generate those messages.

Timers. A protocol may use simple timers that can be reset, and expire after a set time. They can be used by the handler, to time delays or the duration of different modes of operation.

Verification. Since the model will be used for verification, the models will include parts that are only included for this purpose. This will include variables for bookkeeping and methods that check conditions on existing data structures. For this reason, the routing table of the handler was made global, to give access to verifications methods. Otherwise, the routing tables could be a local variable of the corresponding handler. The verification part of the model also often includes a test automaton, which may insert messages, change the topology, and record progress in response to certain events.

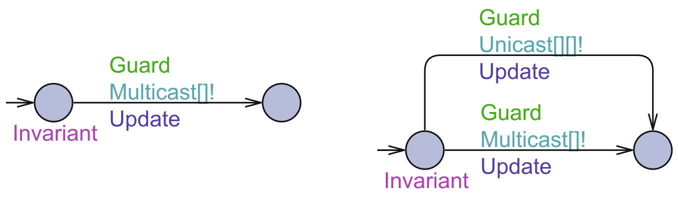


Fig. 3. Unicast and broadcast synchronisation.

The section omits the discussion of type definitions and constants that are used throughout the model. The next section will provide more detail on the various components of the Uppaal model.

4 Generic Uppaal Framework

Our framework consists of global declarations which are global in the system and accessible/updatable by all automata as well as local declarations that are exclusive to each automaton, i.e., these declarations can be accessed and updated only by the automaton itself. There are in total six templates for automata in our framework; four of them are used for modelling protocols and two are concerned with verification. The models are adaptable to protocol specifications, and they are available at http://users.abo.fi/mokamali/FACS2018.

4.1 Communication

To facilitate communication between network nodes in the model, there are a number of global declarations and type definitions. The number of nodes is `const int N`. Addresses of nodes are of type `typedef int[0, N - 1] IP`.

Communication can take place via unicast or multicast messages. The model includes the following channels:

```
broadcast chan  unicast[N][N];
broadcast chan multicast[N];
urgent broadcast chan tau[N];
broadcast chan  newpkt[N];
```

The `tau` channel is used to have internal transitions take place as soon as enabled. They are not used for synchronisation. The `newpkt` channel is used to insert a new packet at a given node.

A protocol must define for each type of message the message format. The reference implementation provides example for packets, route request messages, route reply messages, route error messages, and control messages, also known as TC messages. The format of a TC message, for example, is defined as:

```
typedef struct {
  IP oip;   //originator IP
  int hops; //hops
  TTLT ttl; //time-to-live
  IP sip;   //sender IP
  SQN osn;  //originator sequence number
} TCMSG;
```

The model will include a similar type definition for all types of messages. To make the treatment of message uniform we then define a generic message type as follows:

```
typedef struct {
    MSGTYPE msgtype; //Type of message
    TCMSG tc;        //TC message
    PACKET packet;   //Packet
    RREQMSG rreq;    //Route request msg
    RREPMSG rrep;    //Unicast route reply msg
    RERRMSG rerr;    //Route error msg
} MSG;
```

The field `msgtype` is an index into which type of message is being sent; only the corresponding field should be set. This construction is a work-around for not having *union types* in the Uppaal language.

Each type of message also comes with functions that generate a message of that type. They will be used for convenience and succinctness in the model. It also includes a global variable `MSG msgglobal`, which a sender copies into, and recipients copy from.

4.2 Topology

The network topology is defined by an adjacency matrix `topology[N][N]` with boolean type showing the directed connectivity between nodes, i.e., element 1 in the matrix shows that two nodes are directly connected and 0 indicates that two nodes are not connected directly, however they may be connected via some intermediate nodes. The connectivity between nodes is modelled by function `bool isconnected(IP i, IP j)`, and links can be dropped by calling function `void drop (IP i,IP j)`.

While protocols have to deal with mobility, the mobility models themselves are outside of the scope of this paper and these routing protocols. The processes that establish or delete links – and whether these processes are non-deterministic, stochastic, or probabilistic – are not part of the protocols themselves. The reference model includes a simple model `TopologyChanger` that drops between randomly selected nodes at a rate of 1 : 10. More elaborate models for changing topologies can be found in [10].

We should add here that even if we define a topology matrix to show the direct connectivity between nodes, the network is still wireless. It means that network nodes are not aware of each other before receiving the control messages, and they realise the connectivity only after they receive/process control messages from their neighbours.

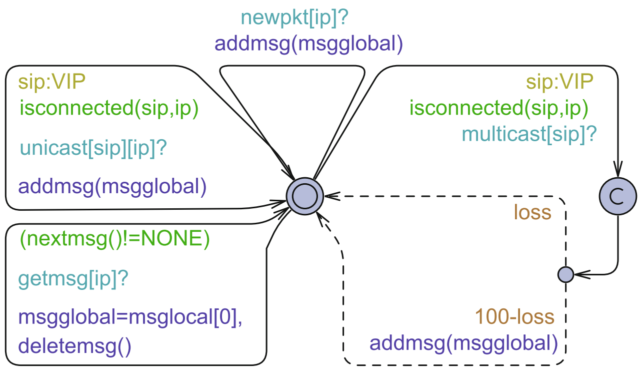


Fig. 4. *Queue* automaton.

4.3 Node

The model for a node using a reactive protocol comprises of two automata, the *Queue* and the *Handler*, proactive protocols also include a `Generator`. If the protocol uses timers, the model includes a fourth automaton to manage the timers. The generic model defines a number of global variables and channels to facilitate synchronisation between these parts. For instance, channel `imsg[N]` is an urgent channel which is used for synchronisation between *Handler* and *Queue* automata.

Queue. The template of the queue defines a number of local constants and variables to manage the stored messages. The most important variable is an array `MSG msglocal[QLength]`. The reference model includes methods `void addmsg(MSG msg)` and `void deletemsg()` to add or delete messages from the queue. For synchronisation with the *Handler* the model includes a global variable `bool isMsgInQ[N]` to encode whether a queue contains at least one message.

The automaton for the queue has essentially one control location, as depicted in Fig.4. It has one self-loop for unicast messages, and one loop for multicast messages that can be received, and one for new packets that are inserted by the tester. The latter loop includes a probabilistic choice to lose the message with probability of `loss`. The automaton also includes a loop, labelled `getmsg?`, for the handler to request the first element of the queue.

Handler (Reactive/Proactive). Nodes have routing tables that store information about other nodes of the network which are empty (initialised to 0 at the beginning) and may be updated when they receive control messages from their neighbour nodes (conditions on when to update routing tables can be specific to each protocol). The reference implementation defines an entry to the routing table as:

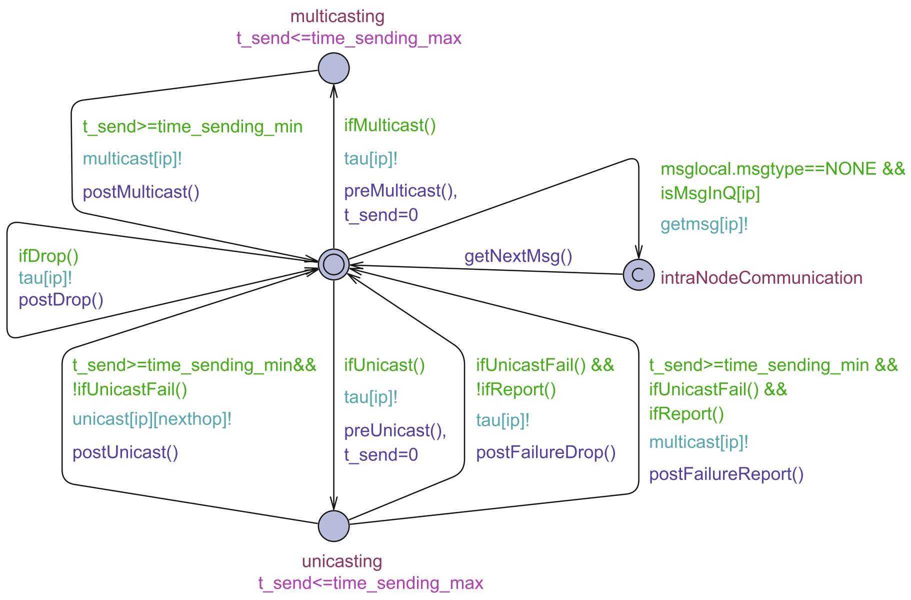


Fig. 5. *Handler* automaton.

```
typedef struct
{ IP dip;
  SQN dsn;
  int hops;
  IP nhop;
} rtentry;
```

The routing tables are then defined as `rtentry art[N][N]`. This is a global variable to allow the *Generator* and verification part read access. Protocols may define additional data structures, for example for error handling.

The handler has a main control location, as depicted in Fig. 5. It includes one loop for multicast messages. The transition to location `multicasting` prepares the message, and waiting in that location up to the permitted amount of time models the message delay, then the transition back to the main location, actually copies the message to the global variable `msgglobal`, for the queue of the receivers to read. The loop for unicast messages has a similar setup, except that includes the option of failure, as in Fig. 3. One option is to drop the message, the other is to multicast an error message. The model also includes for each message type a loop that drops the message, and a loop that requests a new message from the queue.

Most routing protocols use sequence numbers to keep track of newly received information. The reactive version of the handler includes the sequence number, in the proactive version this is a task for the *Generator*.

Generator (Proactive). The task of the *Generator* in a proactive protocol (time-dependent) is to create messages at regular intervals based on the protocol specification. There may be more than one generator, generating more than one type of message. The *Generator* will use a number of clocks to time the generation of those messages as well as to model the sending time of messages (message delays).

Timers (Optional). The protocol may include a fixed number of timers. The model includes as global variables the number of timers, the threshold for each timer, and boolean flags for restart, and to notify that a timer has expired. The automaton managing all timers of one node, has an array of clocks. Once the handler sets the restart flag for a timer to true, the timer automaton will reset a corresponding clock. This transition is urgent, ensuring that no time expires between setting the restart flag, and resetting the corresponding clock. After the threshold duration of the timer, the flag for an expired timer will be set.

4.4 Verification

The model includes for verification purposes a number of side-effect free functions, that can be used by the properties. This includes function to count the number of delivered packets, and how many routes have been established.

For verification the model also includes an automaton *Tester*. The reference model includes an automaton that injects one new packet after about 50 rounds of communication, after which it proceeds to location `final`.

The reference implementation focuses on three main properties, namely for route establishment, network knowledge and packet delivery. These properties verify the core functionality of protocols, e.g., routing data packets.

Route Establishment. The reference model includes the following property for route establishment:

```
Pr[<=1000](<>(route_establishment(OIP1,DIP1)))
```

The function `route_establishment()` returns true if the node `OIP1`, the source node, has the information about the destination node `DIP1` to later send data packet to the destination. The property computes the probability that the function `route_establishment()` returns true in less than or equal to 1000 time units (this value can be altered based on the system requirements).

Network Knowledge. The reference model includes the following property for the network knowledge:

```
E[<=1000;100](max:total_knowledge())
```

This property computes the expected number of connections that have been discovered by time 1000. The function `total_knowledge()` counts for how many originator/destination pairs a route is known.

Packet Delivery. The property for packet delivery is as follows:

```
E[<=1000;100](max:packet_delivered())
```

Packet delivery property shows the number of data packets being delivered at their destinations. The property returns the expected value during 1000 time units by 100 runs. Function `packet_delivered()` is used to count the number of delivered packets.

5 Experiments

We model three well-known routing protocols of WMNs, namely BATMAN, OLSR, AODVv2, to show the reusability and adaptability of our framework. We also verify these protocols for the three different properties discussed in the last section. Each of these is considered for the following scenarios: (1) 0% message loss and no link failures, (2) 80% message loss and no link failures, and (3) 0% message loss and possible link failures.

For all of our experiments, we inject one packet to an arbitrary source to be delivered to an arbitrary destination. We consider networks in a grid, a linear, a fully connected or a ring topology consisting of 9 nodes, as our framework is independent of the number of nodes in the network. It means that number of nodes in the network is adjustable (larger networks are allowed to have) as long as the tool can manage the number of states in the system (state space should be manageable by the tool). A detailed study of these protocols and verification of the models for all possible combinations and/or larger networks are out of the scope of this paper; we only illustrate the applicability and power of the proposed framework.

The automaton `TopologyChanger` is included in models that exhibits link failure. The rate of failure can be simply adjusted based on the protocol specification. We set this value as rate 1 : 10 for all of our models and experiments, e.g., BATMAN, OLSR, AODVv2 models. As none of the protocols that we considered for this study uses timers, we did not have to include the corresponding automaton.

We conduct our experiments using the following set-up: (i) 3.2 GHz Intel Core i5, with 8GB memory, running the Mac OS X 10.11.6 "El Capitan" operating system; (ii) Uppaal SMC model-checker 64-bit version 4.1.20. In Uppaal SMC, two main statistical parameters α and ϵ, in the interval $[0, 1]$, must be fixed by the user. These parameters indicate the probability of *false negatives* and probabilistic *uncertainty*, respectively. In our experiments, these values, i.e., false negatives (α) and probabilistic uncertainty (ϵ) are both set to 0.05, leading to a confidence level of 95%.

5.1 Better Approach to Mobile Ad-hoc Networking (BATMAN)

BATMAN [19] is a proactive protocol used in WMNs. It decentralises route information, i.e., no node has all the data. Each node only maintains information about the possible best next hop. The protocol has two main aims: first, it discovers all bidirectional links and then identifies the best next hop neighbour for all the other nodes in the network. To provide this information, each node broadcasts originator messages (OGMs) through the network at a regular interval. A node keeps track of the information about other known nodes, stored in the node's routing table. When a node receives a message from its neighbours, it updates this information.

Since BATMAN is a proactive protocol, we include a *Generator* for creating OGMs in addition to the *Handler* and the *Queue*. We adapted our framework based on the model of [4]. Table 1 shows the result of our verification for different topologies. We ran the same experiments for the original model of [4] and we got similar results as we got for our adjusted BATMAN model.

Results show that in case of reliable communication (0% loss), the source nodes can find a path to the destination of the injected packet with the probability in the interval [0.90–1.00] and the routing tables are all populated by periodic exchanging of control messages as they were expected by the specification. The injected packet is delivered at the destination in all types of topologies. When message loss increases to 80%, these values may decrease. The same happens also in case of possible link failures. The verification of the route establishment property (probability estimation) takes on average about 2 s whereas the verification for calculating the number of delivered packets and routing table entries (expected value evaluation) takes on average about 215 s (calculating the expected values is more time-consuming compared to estimating the probability).

5.2 Optimised Link State Routing (OLSR)

OLSR[5], a proactive protocol used in WMNs, bears the benefit of having routes to different destinations available to be used whenever needed. This is done by exchanging control messages, namely HELLO and Topology Control (TC), periodically through the network. Receiving nodes update their routing tables based on the information in the messages so that when a packet to be destined to some destination is injected, it can find the path in routing tables.

OLSR differs from other proactive protocols in the way that it minimises flooding of control messages by selecting so-called Multipoint Relays (MPRs). Informally, an MPR takes over the communication for a set of nodes that are one-hop neighbours of this node; these one-hop neighbours receive all the routing information from the MRPs and hence do not need to send and receive routing information from other parts of the network.

Our model for each node includes in addition to the routing table also data structures to manage the selection of MPRs. The model includes two *Generators*, one for HELLO and one for TC messages, the *Handler* and the *Queue*. We adapted our framework based on the model of [15] and verified our generic properties. Table 2 shows the result of our verification for different topologies. We ran the same experiments for the original model of [15] and we got similar results as we got for our adjusted OLSR model.

Table 1. BATMAN verification results

	Route establishment			Network knowledge			Packet delivery		
	0% loss	80% loss	link failure	0% loss	80% loss	link failure	0% loss	80% loss	link failure
Grid	0.90 − 1.00	0.00 − 0.10	0.88 − 0.98	72	7	70	1	0	0.4
Linear	0.90 − 1.00	0.00 − 0.10	0.00 − 0.10	72	4	36	1	0	0
Fully connected	0.90 − 1.00	0.25 − 0.35	0.90 − 1.00	72	22	72	1	0.1	0.8
Ring	0.90 − 1.00	0.24 − 0.34	0.85 − 0.95	72	4	58	1	0.1	0.6

Table 2. OLSR verification results

	Route establishment			Network knowledge			Packet delivery		
	0% loss	80% loss	link failure	0% loss	80% loss	link failure	0% loss	80% loss	link failure
Grid	0.90 − 1.00	0.33 − 0.43	0.90 − 1.00	72	65	70	1	0	0.3
Linear	0.90 − 1.00	0.00 − 0.10	0.18 − 0.28	72	34	50	1	0	0
Fully connected	0.90 − 1.00	0.90 − 1.00	0.89 − 0.99	72	72	72	1	1	0.5
Ring	0.90 − 1.00	0.90 − 1.00	0.89 − 0.99	72	43	63	1	1	0.6

Table 3. AODVv2 verification results

	Route establishment			Network knowledge			Packet delivery		
	0% loss	80% loss	link failure	0% loss	80% loss	link failure	0% loss	80% loss	link failure
Grid	0.90 − 1.00	0.00 − 0.10	0.14 − 0.24	28	3	21	1	0	0.3
Linear	0.90 − 1.00	0.00 − 0.10	0.00 − 0.10	72	1	8	1	0.1	0
Fully connected	0.90 − 1.00	0.16 − 0.26	0.90 − 1.00	9	21	11	1	0.1	0.8
Ring	0.90 − 1.00	0.11 − 0.21	0.79 − 0.89	30	2	10	1	0	0.7

Results indicate that in case of reliable communication (0% loss), the source nodes can find a path to the destination of the injected packet with the probability in the interval [0.90–1.00] and the routing tables are all populated by periodic exchanging of control messages as they were expected by the specification. The injected packet is delivered at the destination in all types of topologies. When message loss increases to 80%, these values may decrease. The same happens also in case of possible link failures. The verification regarding route establishment property (probability estimation) takes on average about 2 s whereas the verification for calculating the number of delivered packets and routing table entries (expected value evaluation) takes on average about 185 s (calculating the expected values is more time-consuming compared to estimating the probability).

5.3 Ad-Hoc On-Demand Distance Vector Version2 (AODVv2)

AODVv2 [21], a reactive protocol for WMNs, behaves on-demand. This means that it tries to find a route to the destination when a packet is injected into the network. The protocol initiates RREQ message and the receiving nodes update their routing tables and possibly rebroadcast the message until the RREQ is received by its destination. Then the destination sends a RREP message back to the source of the RREQ. In this way, a path from the source to the destination is created and the packet can be forwarded via that path. AODVv2 will report failure of links by multicasting RERR messages.

The model of AODVv2 protocol contains only models for the *Handler* and the *Queue*. As a reactive protocol, it does not need a generator. Compared to the other two models, AODVv2 has more message types, as it includes error reporting. This also means that in addition to routing information, each node maintains information of routing errors. We adapted our framework based on the model of [16] and verified our generic properties. Table 3 shows the result of our verification for different topologies. We ran the same experiments for the original model of [16] and we got similar results as we got for our adjusted AODVv2 model which shows the adaptability and reusability of our framework.

Results show that in case of reliable communication (0% loss), the source nodes can find a path to the destination of the injected packet with the probability in the interval [0.90–1.00]. Routing tables are partially populated by periodic exchanging of control messages as they were expected by the specification (reactive protocols find paths to destinations on-demand, so it is expectable that not all tables are updated). However in the linear topology, all routing tables are updated due to the path accumulation feature of AODVv2, meaning that whenever a control message travels via more than one node, information about all intermediate nodes is accumulated in the message and then is distributed to its recipients.

The injected packet is delivered at the destination in all types of topologies in case of reliable communication (0% message loss). When message loss increases to 80%, these values may decrease. The same happens also in case of possible link failures. The verification regarding route establishment property (probability estimation) takes on average about 2 s whereas the verification for calculating the number of delivered packets and routing table entries (expected value evaluation) takes on average about 15 s (calculating the expected values is more time-consuming compared to estimating the probability).

Evaluating the expected values for AODVv2 takes less time due to the reactive characteristic of AODVv2, meaning that since AODVv2 broadcasts control messages on demand it has less number of states compared to BATMAN and OLSR that broadcast control messages periodically which decreases the time spent for verification. As all the three protocols are modelled applying our framework, it is possible to easily compare the protocols w.r.t. the properties and verification time.

5.4 Discussion on BATMAN, OLSR and AODVv2 Models

Here, we discuss how much our framework needs the interaction from the modeller to be adjusted based on the protocol specification. In other words, how much the three case studies and our framework have in common and how much they are different. The general structure of our six automata (locations and transitions of the automata), i.e., *Handler*, *Queue*, *Generator*, *Timer*, *TopologyChanger* and *Tester*, and their synchronisation remain unchanged and only some declaration (code fragments) of the automata may need to be modified/added based on the specification of the protocol.

- Communication: format of each message has been separately modified using *typedef struct* (based on BATMAB, OLSR and AODVv2 specifications) and later is added in our generic message *MSG*. The *IP* address of nodes, *SQN* sequence numbers, channels, etc are borrowed from the framework.
- Topology: connectivity function, network topology and *TopologyChanger* automata remain unchanged. We have only borrowed them from our framework.
- Node: the *Queue* and the *Timer* automata remain unchanged and they have been only imported and used. Function *createMSG* in the *Generator* declaration which is applicable only for BATMAN and OLSR, has been modified based on the specification (as mentioned earlier, the format of messages for different protocols are unique to the protocol and must be changed based on the specification). The interval for sending periodic messages is a parameter of each protocol and should be set in the declarations.

  The *Handler* needs more interactions from the modeller when modifying the local declaration of the automaton. This is the case due to different behaviour of protocols, e.g., when to update a routing table, when to process/drop a message, when to multicast a message, etc. For instance, BATMAN protocol has a specific procedure for storing sequence numbers which is unique to this protocol, OLSR has a specific procedure for determining MPRs, and AODVv2 has a specific procedure for accumulating paths. These specific features need to be separately modelled for each protocol and our framework only supports the standardise behaviour of routing protocols which were discussed earlier. Our models move much of the logic to functions inside the model in order to have the core of the protocol as code fragments in the model which makes the modelling task easier.
- Verification: the three system requirements (properties) and their corresponding functions have also been imported without any modifications. The *Tester* automaton injects the packet in accordance to the category of the protocol; reactive or proactive protocol. If the protocol is proactive (BATMAN and OLSR), the *Tester* automaton injects the packet after routes are discovered; and if the protocol is reactive (AODVv2), the *Tester* injects the packet for route discovery process and the routes are discovered later after packet injection. It means that only the time interval that the *Tester* transition is enabled differs for reactive and proactive protocols.

6 Related Work

Formal modelling and analysis of the WMNs and Mobile Ad-hoc Networks (MANETs) and their routing protocols is among challenging tasks, and formal verification of such systems has attracted the attention from formal methods community [3,11,17]. Fehnker et al. [8] applied the Uppaal model checker [2] for analysing qualitative properties of the AODV protocol in all network topologies with five nodes. Kamali et al. [15] focused on formal modelling and verifying OLSR protocol in network topologies with five nodes. They have also applied

Event-B to model OLSR and have analysed this protocol in large networks (no size barrier w.r.t. the size of the network) [14]. Chaudhary et al. [4] formally modelled BATMAN routing protocol using Uppaal model checker revealing several ambiguities in the RFC. They verified their model for loop-freedom, bidirectional link discovery, and route-discovery. Fehnker et al. [9] modelled and verified LMAC protocol of wireless sensor networks applying Uppaal. Their study was carried out to detect and resolve collision in networks consisting of four and five nodes.

There are several studies using (statistical) model checking to analyse WMN and MANET routing protocols. Höfner and McIver [13] made a comparison of the AODV and DYMO protocols on arbitrary networks up to five nodes considering perfect communication among nodes, applying the Uppaal SMC model checker. Their analysis shows that DYMO has worsened performance compared to AODV. Dal Corso et al. [6] studied the extended and generalised work done by [13] to 4×3 grids with lossy communication. They showed contrary results, indicating that DYMO is performing better compared to AODV. Kamali et al. [16] investigated and compared the performance and looping property of the most recent version of AODV protocol [21] with DYMO on 3×3 grids. Their results indicate that the more recent version of AODV pays the price of degraded performance compared to DYMO to remain loop-free.

There are other studies providing formal frameworks for modelling and verifying MANETs. Liu et al. [18] presented a formal modelling framework for MANETs consisting of several mobility models together with wireless communication applying Real-Time Maude [20]. They analysed the AODV protocol using their framework and their mobility models. Their framework mainly focuses only on integrating a number of mobility models together with wireless communication. Yousefi et al. [22] have modelled MANETs using the extension of an actor-based modelling language bRebeca. They provided a framework to detect malfunctioning of MANET protocols, addressing local broadcast and topology changes. They have modelled the core functionality of AODV protocol and found some malfunctioning of this protocol (loop existence).

Our work differs from the other previous works in the sense that it models the core functionality of WMN routing protocols, considering wireless communication, topology, message loss, message queuing, link failure, etc. It is also possible to model timing aspects of protocols (both reactive and proactive) and to allow probabilities to have both qualitative and quantitative analysis.

In addition, networks of timed automata as the specification language used for introducing our generic framework (the main common components of routing protocols) are more manageable to alter based on the protocols specifications. It means that adapting our framework allows protocol designers to have an insight of the system before the deployment since timed automata is an easy-to-understand specification language and Uppaal SMC simulator provides the means to validate the system which later can be also used for verification. Protocol designers can simply modify the C-like code in the declarations based on the protocol specification where the general structure of networks of different automata remains unchanged.

7 Conclusion

This paper presented an adaptive, generic and reusable framework as well as crucial generic properties to model and verify WMN routing protocols. This framework uses Uppaal SMC to capture timing aspect of protocols, communication between nodes, and probabilities to model message loss, link breakage, etc.

This paper discussed the general structure of Uppaal models, and how this influences the design of models for network routing protocols. It described how to build reusable components within the constraints imposed by the Uppaal modelling language. It identified the main components that routing protocols have in common, and how to map them to data structures, processes, channels, and timed automata in the Uppaal language. We demonstrated the applicability of the approach by implementing three different protocols in this framework: AODVv2, OLSR and BATMAN.

One of the characteristics of these models is that they move much of the logic to functions inside of the Uppaal model. They rely less on the subtle interplay of channels, urgent locations, or committed locations. Instead, they standardise proven patterns that have been used in the community to model routing protocols. This also means that the core of the protocol resides as code fragments in the model, and becomes available to be standard code reviewing practices.

An observation that was made is that Uppaal as modelling language would benefit if it would adopt more mechanisms to structure code. It would be beneficial if the model could reflect that a number of templates share access to data structures to the exclusion of others. Often the workaround for sharing information is to make data global, without mechanisms to enforce its consistent use. Furthermore, code that is included for verification is currently scattered across the model. It might be worth to consider verification as a cross-cutting concern, similarly to how these are dealt with in aspect-oriented programming.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994)
2. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30080-9_7
3. Bhargavan, K., Obradovic, D., Gunter, C.A.: Formal verification of standards for distance vector routing protocols. J. ACM **49**(4), 538–576 (2002)
4. Chaudhary, K., Fehnker, A., Mehta, V.: Modelling, verification, and comparative performance analysis of the B.A.T.M.A.N. protocol. In: Hermanns, H., Höfner, P. (eds.) MARS 2017, vol. 244, pp. 53–65 (2017)

5. Clausen, T., Jacquet, P.: Optimized link state routing protocol (OLSR). RFC3626 (2003). http://www.ietf.org/rfc/rfc3626
6. Dal Corso, A., Macedonio, D., Merro, M.: Statistical model checking of ad hoc routing protocols in lossy grid networks. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 112–126. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17524-9_9
7. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B.: Uppaal SMC tutorial. STTT **17**(4), 397–415 (2015)
8. Fehnker, A., van Glabbeek, R., Höfner, P., McIver, A., Portmann, M., Tan, W.L.: Automated analysis of AODV using UPPAAL. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 173–187. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_13
9. Fehnker, A., van Hoesel, L., Mader, A.: Modelling and verification of the LMAC protocol for wireless sensor networks. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 253–272. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73210-5_14
10. Fehnker, A., Höfner, P., Kamali, M., Mehta, V.: Topology-based mobility models for wireless networks. In: Joshi, K., Siegle, M., Stoelinga, M., D'Argenio, P.R. (eds.) QEST 2013. LNCS, vol. 8054, pp. 389–404. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40196-1_32
11. van Glabbeek, R., Höfner, P., Portmann, M., Tan, W.L.: Modelling and verifying the AODV routing protocol. Distrib. Comput. **29**(4), 279–315 (2016)
12. Hérault, T., Lassaigne, R., Magniette, F., Peyronnet, S.: Approximate probabilistic model checking. In: Steffen, B., Levi, G. (eds.) Verification, Model Checking, and Abstract Interpretation, pp. 73–84. Springer, Berlin (2004)
13. Höfner, P., McIver, A.: Statistical model checking of wireless mesh routing protocols. In: Brat, G., Rungta, N., Venet, A. (eds.) NFM 2013. LNCS, vol. 7871, pp. 322–336. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38088-4_22
14. Kamali, M., Petre, L.: Modelling link state routing in event-B. In: Wang, H., Mokhtari, M. (eds.) ICECCS 2016, pp. 207–210. IEEE (2016)
15. Kamali, M., Höfner, P., Kamali, M., Petre, L.: Formal analysis of proactive, distributed routing. In: Calinescu, R., Rumpe, B. (eds.) SEFM 2015. LNCS, vol. 9276, pp. 175–189. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22969-0_13
16. Kamali, M., Merro, M., Dal Corso, A.: AODVv2: performance vs. loop freedom. In: Tjoa, A.M., Bellatreche, L., Biffl, S., van Leeuwen, J., Wiedermann, J. (eds.) SOFSEM 2018. LNCS, vol. 10706, pp. 337–350. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73117-9_24
17. Kamali, M., Petre, L.: Improved recovery for proactive, distributed routing. In: ICECCS 2015, pp. 178–181. IEEE (2015)
18. Liu, S., Ölveczky, P.C., Meseguer, J.: A framework for mobile ad hoc networks in real-time maude. In: Escobar, S. (ed.) WRLA 2014. LNCS, vol. 8663, pp. 162–177. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12904-4_9
19. Neumann, A., Aichele, C., Lindner, M., Wunderlich, S.: Better approach to mobile ad-hoc networking (BATMAN). Internet draft00 (2008). https://tools.ietf.org/html/draft-wunderlich-openmesh-manet-routing-00
20. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. High.-Order Symb. Comput. **20**(1), 161–196 (2007)

21. Perkins, C., Stan, R., Dowdell, J., Steenbrink, L., Mercieca, V.: Ad hoc on-demand distance vector version 2 (AODVv2) routing. Internet Draft 16 (2016). https://datatracker.ietf.org/doc/draft-ietf-manet-aodvv2
22. Yousefi, B., Ghassemi, F., Khosravi, R.: Modeling and efficient verification of wireless ad hoc networks. Form. Asp. Comput. **29**(6), 1051–1086 (2017)