Automated Analysis of AODV using UPPAAL*

Ansgar Fehnker[1,4], Rob van Glabbeek[1,4], Peter Höfner[1,4], Annabelle McIver[2,1], Marius Portmann[1,3] and Wee Lum Tan[1,3]

[1] NICTA
[2] Department of Computing, Macquarie University
[3] School of ITEE, The University of Queensland
[4] Computer Science and Engineering, University of New South Wales

Abstract. This paper describes an automated, formal and rigorous analysis of the Ad hoc On-Demand Distance Vector (AODV) routing protocol, a popular protocol used in wireless mesh networks.

We give a brief overview of a model of AODV implemented in the UPPAAL model checker. It is derived from a process-algebraic model which reflects precisely the intention of AODV and accurately captures the protocol specification. Furthermore, we describe experiments carried out to explore AODV's behaviour in all network topologies up to 5 nodes. We were able to automatically locate problematic and undesirable behaviours. This is in particular useful to discover protocol limitations and to develop improved variants. This use of model checking as a diagnostic tool complements other formal-methods-based protocol modelling and verification techniques, such as process algebra.

1 Introduction

Route finding and maintenance are critical for the performance of networked systems, particularly when mobility can lead to highly dynamic and unpredictable environments; such operating contexts are typical in wireless mesh networks (WMNs). Hence correctness and good performance are strong requirements of routing algorithms. The Ad hoc On-Demand Distance Vector (AODV) routing protocol [11] is a widely used routing protocol designed for WMNs and mobile ad hoc networks (MANETs). It is one of the four protocols defined in an RFC (Request for Comments) document by the IETF MANET working group. AODV also forms the basis of new WMN routing protocols, like the upcoming IEEE 802.11s wireless mesh network standard [7].

Usually, routing protocols are optimised to achieve key objectives such as providing self-organising capability, overall reliability and performance in typical network scenarios. Additionally, it is important to guarantee protocol properties such as loop freedom for *all* scenarios, including non-typical, unanticipated ones. This is particularly relevant for highly dynamic MANETs and WMNs.

The traditional approaches for the analysis of MANET and WMN routing protocols are simulation and test-bed experiments. While these are important

* First steps towards this analysis appeared in [5].

and valid methods for protocol evaluation, there are limitations: they are resource intensive and time-consuming. The challenges of extensive experimental evaluation are illustrated by recent discoveries of limitations of protocols that have been under intense scrutiny over many years. An example is [9].

We believe that formal methods in general and model checking in particular can help in this regard. Model checking is a powerful method that can be used to validate key correctness properties in finite representations of a formal system model. In the case that a property is found not to hold, the model checker produces evidence for the fault in the form of a "counter-example" summarising the circumstances leading to it. Such diagnostic information provides important insights into the cause and correction of these failures.

In [4], we specified the AODV routing protocol in the process algebra AWN. The specification follows well-known programming constructs and lends itself well for comparison with the original specification of the protocol in English. Based on such a comparison we believe that the AWN model provides a complete and accurate formal specification of the core functionality of AODV. In developing the formal specification, we discovered a number of ambiguities in the IETF RFC [11]. Our process algebraic formalisation captures these by several interpretations, each with slightly different AWN code.

In this paper we follow an interpretation of the RFC, which we believe to be the closest to the spirit of the AODV routing protocol. We show how to obtain executable versions of this AWN specification, in the language of the UPPAAL model checker [1,8]. By deriving the UPPAAL model from the AWN model, the accuracy of the AWN model is transferred to the UPPAAL model.

The executable UPPAAL model is used to confirm and discover the presence of undesirable behaviour. We check important properties against all topologies of up to 5 nodes, which also includes dynamic topologies with one link going up or down. This exhaustive search confirmed known and revealed new problems of AODV, and let us quantify in how many topologies a particular error can occur. Subsequently, the same experiments for modifications of AODV showed the proposed modifications can all but eliminate certain problems for static topologies, and significantly reduce them for dynamic topologies. The automated analysis of routing protocols presented in this paper combined with formal reasoning in AWN provides a powerful tool for the development and rigorous evaluation of new protocols and variations, and improvements of existing ones.

2 Ad hoc On-Demand Distance Vector Routing Protocol

2.1 The Basic routine

AODV [11] is a widely used routing protocol designed for WMNs and MANETs. It is a reactive routing protocol, where the route between a source and a destination node is established on an on-demand basis. A route discovery process is initiated when a source node s has data to send to a destination node d, but has no valid corresponding routing table entry. In this case, node s broadcasts a route

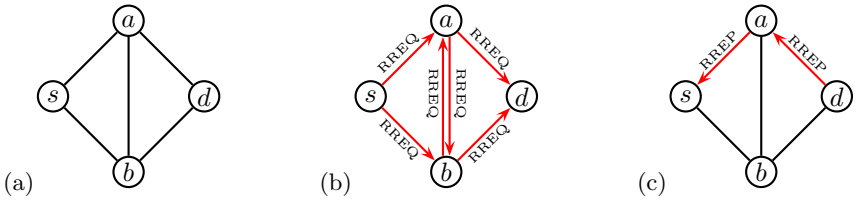

Fig. 1. Example network topology

request (RREQ) message in the network. The RREQ message is re-broadcast and forwarded by other intermediate nodes in the network, until it reaches the destination node d (or an intermediate node that has a valid route to node d). Every node that receives the RREQ message will create a routing table entry to establish a *reverse route* back to node s. In response to the RREQ message, the destination node d (or an intermediate node that has a valid route to node d) unicasts a route reply (RREP) message back along the previously established reverse route. At the end of this route discovery process, an end-to-end route between the source node s and destination node d is established. Usually, all nodes on this route have a routing table entry to both the source node s and destination node d. An example topology, indicating which nodes are in transmission range of each other, as well as the flow of RREQ and RREP messages, is given in Figure 1. In the event of link and route breaks, AODV uses route error (RERR) messages to inform affected nodes. Sequence numbers are another important aspect of AODV, and are used to indicate the freshness of routing table entries for the purpose of preventing routing loops.

2.2 Process Algebraic Model of AODV

The process algebra AWN [4] has been developed specifically for modelling WMN routing protocols. It is designed in a way to be easily readable and treats three necessary features of WMNs routing protocols: *data structures*, *local broadcast*, and *conditional unicast*. Data structures are used to model routing tables etc.; local broadcast models message sending to *all* directly connected nodes; and conditional unicast models the message sending to one particular node and chooses a continuation process dependent on whether the message is successfully delivered.

In AWN, delivery of broadcast messages is "guaranteed", i.e., they are received by any neighbour that is directly connected. The abstraction to a guaranteed broadcast enables us to interpret a failure of message delivery (under assumptions on the network topology) as an imperfection in the protocol, rather than as a consequence of unreliable communication. Section 4.3, for example, describes a simple network topology and a scenario for which AODV fails to discover a route, even if broadcast is guaranteed. The failure is a shortcoming of the protocol itself, and cannot be excused by unreliable communication.

Conditional unicast models an abstraction of an acknowledgment-of-receipt mechanism that is typical for unicast communication but absent in broadcast communication, as implemented by the link layer of relevant wireless standards

such as IEEE 802.11. The AWN model captures the bifurcation depending on the success of the unicast, while abstracting from all implementation details.

In [4], we used AWN to model AODV according to the IETF RFC [11]. The model captures all core functionalities as well as the interface to higher protocol layers via the injection and delivery of application layer data, and the forwarding of data packets at intermediate nodes. Although the latter is not part of the AODV protocol specification, it is necessary for a practical model of any reactive routing protocol where protocol activity is triggered via the sending and forwarding of data packets. In addition, our model contains neither ambiguities nor contradictions, both of which are often present in specifications written in natural languages, such as in the RFC3561 (see e.g. [4]).

The AWN model of AODV contains a main process, called `AODV`, for every node of the network, which handles messages received and calls the appropriate process to handle them. The process also handles the forwarding of any queued data packet if a valid route to its destination is known. Four other processes handle one particular message type each, like RREQ. The network as a whole is modelled as a parallel composition of these processes. Special primitives allow us to express whether two nodes are connected. Full details of the process algebra description on which our UPPAAL model is based can be found in [4].

3 Modelling AODV in UPPAAL

UPPAAL [1,8] is an established model checker for *networks of timed automata*, used in particular for protocol verification. We use UPPAAL for the following reasons: (1) UPPAAL provides two synchronisation mechanisms—binary and broadcast synchronisation, which translate to uni- and broadcast communication; (2) it provides common data structures, such as arrays and structs, and a C-like programming language to define updates on these data structures; (3) in the future, AWN (and therefore also our models) will be extended with time and probability—UPPAAL provides mechanisms and tools for both.

Our process-algebraic model of AODV has been used to prove essential properties, such as loop freedom for popular interpretations of [11]—independent of a particular topology. The UPPAAL model is derived from the AWN specification that comes closest to the spirit of the AODV routing protocol.

Section 3.2 will explain the translation and the simplifying assumptions in detail.

3.1 UPPAAL Automata

Since our models do not yet use time (or probabilities) they are simply *networks of automata* with guards. The state of the system is determined, in part, by the values of data variables that can be either shared between automata, or local. We assume a data structure with several types, variables ranging over these types, operators and predicates. Common Boolean and arithmetic expressions are used to denote data values and statements about them.

Each automaton is a graph, with locations, and edges between locations. Every edge has a guard, optionally a synchronisation label, and an update. Synchronisation occurs via so-called channels; for each channel a there is one label $a!$ to denote the sender, and $a?$ to denote the receiver. Transitions without labels are internal; all other transitions use one of two types of synchronisation.

In *binary handshake* synchronisation, one automaton having an edge with a label that has the suffix ! synchronises with another automaton with an edge having the same label that has a ?-suffix. These two transitions synchronise when both guards are true in the current state, and only then. When the transition is taken both locations change, and the updates will be applied to the state variables; first the updates on the !-edge, then the updates on the ?-edge. If there is more than one possible pair, then the transition is selected non-deterministically.

In *broadcast* synchronisation, one automaton with a !-labelled edge synchronises with a set of other automata that all have an edge with a matching ?-label. The initiating automaton can change its location, and apply its update, if the guard on its edge evaluates to true. It does not require a second synchronising automaton. Automata with a matching ?-labelled edge have to synchronise if their guard is currently true. They change their location and update the state. The automaton with the !-edge will update the state first, followed by the other automata in some lexicographic order. If more than one automaton can initiate a transition on an !-edge, the choice will be made non-deterministically.

3.2 From AWN to UPPAAL

Every node in the network is modelled as a single automaton, each having its own data structures such as a routing table and message buffer. The implementation of the data structure defined in AWN is straightforward, since both AWN and UPPAAL allow C-style data structures. A routing table `rt` for example is an array of entries, one entry for every node. An entry is given by the data type

```
typedef struct
{ SQN dsn;     //destination sequence number
  bool flag;   //validity of a routing table entry
  int hops;    //distance (hop count) to the destination
  IP nhop;     //next hop (is 0 if no route)
} rtentry;
```

where `SQN` denotes a data type for sequence numbers and `IP` denotes one for all IP address. In our model, these types are mapped to integers.

The local message buffer is modelled as an array `msglocal`. UPPAAL will warn if during model checking an out-of-bounds error occurs, i.e., if the array was too small. Each message is a struct with fields `msgtype` which can take values `PKT`, `RREQ`, `RREP`, or `RERR`, integer `hops` for the distance from the originator of the message, sequence number `rreqid` to identify a route request, a destination IP `dip`, a destination sequence number `dsn`, an originator IP `oip`, an originator sequence number `osn`, and a sender IP `sip`. The model contains functions `addmsg`, `deletemsg` and `nextmsg`, to add a message, delete a message, or to return the type of the next message in the buffer.

Table 1 Excerpt of AWN spec for AODV. A few cases for RREQ handling.

```
AODV(ip,sn,rt,rreqs,store) =def
 1. /*depending on the message on top of the message queue, the node calls different processes*/
 2. ...
 3. [ msg = rreq(hops, rreqid, dip, dsn, oip, osn, sip) ∧ (oip, rreqid) ∈ rreqs ]
 4.    /*silently ignore RREQ, i.e. do nothing, except update the entry for the sender*/
 5.    ⟦rt := update(rt, (sip, 0, val, 1, sip))⟧ .  /*update the route to sip*/
 6.    AODV(ip,sn,rt,rreqs,store)
 7. + [ msg = rreq(hops, rreqid, dip, dsn, oip, osn, sip) ∧ (oip, rreqid) ∉ rreqs ∧ dip = ip ]
 8.    /*answer the RREQ with a RREP*/
 9.    ⟦rt := update(rt, (oip, osn, val, hops + 1, sip))⟧  /*update the routing table*/
10.    ⟦rreqs := rreqs ∪ {(oip, rreqid)}⟧  /*update the array of already seen RREQ*/
11.    ⟦sn := max(sn, dsn)⟧  /*update the sqn of ip*/
12.    ⟦rt := update(rt, (sip, 0, val, 1, sip))⟧  /*update the route to sip*/
13.    unicast(nhop(rt,oip),rrep(0,dip,sn,oip,ip)) .
14.    AODV(ip,sn,rt,rreqs,store)
15. + [ msg = rreq(hops, rreqid, dip, dsn, oip, osn, sip) ∧(oip, rreqid) ∉ rreqs ∧ dip ≠ ip ∧
        (dip ∉ vD(rt) ∨ sqn(rt,dip) < dsn ∨ sqnf(rt,dip) = unk) ]
16.    /*forward RREQ*/
17.    ⟦rt := update(rt, (oip, osn, val, hops + 1, sip))⟧  /*update routing table*/
18.    ⟦rreqs := rreqs ∪ {(oip, rreqid)}⟧  /*update the array of already seen RREQ*/
19.    ⟦rt := update(rt, (sip, 0, val, 1, sip))⟧  /*update the route to the sender*/
20.    broadcast(rreq(hops + 1,rreqid,dip,max(sqn(rt, dip), dsn),oip,osn,ip)) .
21.    AODV(ip,sn,rt,rreqs,store)
22. + [ rreq(hops, rreqid, dip, dsn, oip, osn, sip) ∧ ... ]
23.    ...
```

Connections between nodes are determined by a *connectivity graph*, which is specified by a Boolean-valued function `isconnected`. This graph presents one particular topology and is not derived from our AWN specification, since the specification is valid for *all* topologies. Communication is modelled as an atomic synchronised transition between a sender, on an !-edge, with a receiver, on a matching ?-edge. The guard of the sender depends on local data, e.g. buffer and routing table, while the guard of the receiver is `isconnected`. This means that in broadcast communication the sender will take the transition regardless of `isconnected`, while disconnected nodes will not synchronise. In unicast communication the transition is blocked if the intended recipient is not connected, but there is a matching broadcast transition that sends an error message in this case. When the transition is taken, the sender copies its message to a global variable `msgglobal`, and the receiver copies it subsequently to its local buffer `msglocal`.

AODV uses unicast for RREP and PKT messages, and broadcast for RERR and RREQ messages. To model unicast, the UPPAAL model has one binary handshake channel for every pair of nodes. For example, `rrep[i][j]` is used for transitions modelling the sending of a route reply from node i to j. To model broadcast, we use one broadcast channel for every node. For example, `rreq[i]` is used for the route requests of node i. To model new packets from i to j, generated by the user layer, the model contains a channel `newpkt[i][j]`.

The AWN model of Table 1 is an excerpt of the AODV specification presented in [4]—the full specification and a detailed explanation can be found there. The excerpt presented here differs slightly from the original model:[1] (1) we abstract

[1] It can be shown that the model presented here behaves identical to the AWN model in [4]; in other words, they are behavioural equivalent.

Table 2 Excerpt of UPPAAL model. A few cases for RREQ handling.

```
1.  ...
2.  aodv -> aodv {
3.  guard nextmsg()==RREQ && rreqs[msglocal[0].oip][msglocal[0].rreqid];
4.  sync  tau[ip]?;
5.  assign sipupdate(), deletemsg();  },
6.  aodv -> aodv {
7.  guard nextmsg()==RREQ&&!rreqs[msglocal[0].oip][msglocal[0].rreqid]&&msglocal[0].dip==ip;
8.  sync  rrep[ip][oipnhop()]!;
9.  assign updatert(msglocal[0].oip,msglocal[0].osn,1,msglocal[0].hops+1,msglocal[0].sip),
10.       rreqs[msglocal[0].oip][msglocal[0].rreqid]=1,
11.       sn=max(sn,msglocal[0].dsn),
12.       sipupdate(),
13.       msgglobal=createrep(0,msglocal[0].dip,sn,msglocal[0].oip,ip), deletemsg();  },
14. aodv -> aodv {
15. guard nextmsg()==RREQ&&!rreqs[msglocal[0].oip][msglocal[0].rreqid]&&msglocal[0].dip!=ip
        && (!rt[msglocal[0].dip].flag || msglocal[0].dsn>rt[msglocal[0].dip].dsn
        || rt[msglocal[0].dip].dsn==0);
16. sync  rreq[ip]!;
17. assign updatert(msglocal[0].oip,msglocal[0].osn,1,msglocal[0].hops+1,msglocal[0].sip),
18.       rreqs[msglocal[0].oip][msglocal[0].rreqid]=1,
19.       sipupdate(),
20.       msgglobal=createreq(msglocal[0].hops+1,msglocal[0].rreqid,msglocal[0].dip,
          max(msglocal[0].dsn, rt[msglocal[0].dip].dsn),msglocal[0].oip,msglocal[0].osn,ip),
21.       deletemsg();  },
22. ...
```

from *precursors*, an additional data structure that is maintained by AODV (2) the model in [4] uses 6 different processes; here processes are inlined into the body of the main AODV process. This reduces the number of processes to one and yields an automaton with one control location; (3) the model in [4] uses nesting of conditions and updates, while this model has been flattened to correspond more closely with the limitations of the UPPAAL syntax—in UPPAAL the *guards* are evaluated before any update, AWN has no such restriction.

Table 1 depicts three of the cases in the AWN model for handling route requests. In each, a condition is checked, the routing tables and local data are updated, and it returns to the main AODV process $\texttt{AODV}(\texttt{ip}, \texttt{sn}, \texttt{rt}, \texttt{rreqs}, \texttt{store})$. Table 2 shows the corresponding edges from the UPPAAL model, one edge for every case. Like the AWN model, which goes from the process `AODV` to `AODV`, the UPPAAL model will go from control location `aodv` to itself (Lines 2, 6 and 14).

Each edge evaluates a guard in Lines 3, 7 and 15 in Table 2. These line numbers, and the line numbers mentioned in the remainder of this section correspond to the same line number in Table 1. Whenever the AWN specification uses set membership ($(\texttt{oip}, \texttt{rreqid}) \in \texttt{rreqs}$), the UPPAAL model uses a 2-dimensional Boolean array `rreqs` to encode membership; whenever the AWN model uses a flag to denote a *known* sequence number ($\texttt{sqnf}(\texttt{rt},\texttt{dip}) = \texttt{unk}$), the UPPAAL model compares with a distinguished value (`rt[msglocal[0].dip].dsn==0`).

Depending on whether a case requires no transmission, unicast, or broadcast, the UPPAAL model synchronises on a `tau`, a binary, or a broadcast channel (Lines 4, 8 and 16). The `tau` channel for internal transitions allows for optimisations; it could have been left empty. We discuss this later in this section.

After synchronisation the state is updated. For all route request messages we update the routing table for the sender `sip` (Lines 5, 12 and 19). The fact that

the message was received means that sender `sip` is one hop away. Except for the first case (Lines 4) the routing table is updated (Lines 9 and 17), and the route request is added to the set of processed route requests (Lines 10 and 18). In case that a node receives a request, and it is the destination, it increments its sequence number, if necessary (Line 11), before it sends a route reply.

The last two steps in the UPPAAL model that complete a transmission first create a new message and copy it to the global variable `msgglobal` (Lines 13 and 20), and then delete the first element of the local message buffer. In the AWN model, these steps are part of the communication primitives.

The full UPPAAL models a node by an automaton with one control location and 26 edges: 19 cases for processing the different routing messages, four cases for receiving routing messages—one case for each type—two cases for sending data packets, and one case for handling new data packets. The case distinction is complete, i.e at least one transition is enabled and process messages if the buffers and queues are not empty.

Both the UPPAAL and the AWN model maintain a FIFO buffer for incoming messages. Any newly generated message only depends on the content of messages previously received. This implies that the timing of internal transitions that discard incoming messages is not relevant for route discovery. The UPPAAL model exploits this fact and assigns a higher priority to internal transitions. To implement priorities we labelled those transitions `tau`. This is is an effective measure to reduce the state space, at the expense that UPPAAL is now unable to check liveness properties; for this paper this is not a limitation, as all properties can be expressed as safety properties.

4 Experiments

Our automated analysis of AODV considers 3 properties that relate to "route discovery" for all topologies up to 5 nodes, with up to one topology change, and scenarios with two new data packets.

4.1 Scenarios and Topologies

The experiments consider scenarios with two initial data packets in networks with up to 5 nodes. Initially all routing tables and buffers are empty. The originator and the destination of the data packets are identified as nodes A, B, or C. The new data packets may arrive as depicted in Figure 2. In the first scenario a packet from A to B is followed by a packet from A to C; in the second a packet from B to A by a packet from C to A; in the third a packet from A to B by a packet from B to C; and in the final scenario a packet from B to C by a packet from A to B. The originator of the first new packet initiates a route discovery process first, the originator of the second non-deterministically after the first. The different scenarios are implemented by a simple automaton, `tester`. Since the different topologies cover all possible permutations, these four

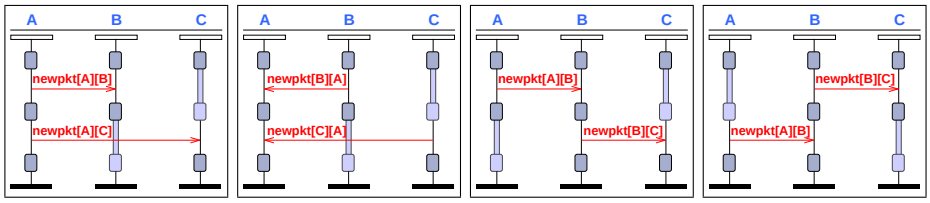

Fig. 2. Sequence charts illustrating four scenarios for initiating two route requests.

scenarios cover all scenarios for injecting two new packets with either different originators or different destinations.

Additional to the nodes A, B and C, we add up to two nodes that may relay messages, but do not create data packets themselves. We consider only topologies in which nodes A, B and C are connected, either directly, or indirectly. This ensures that the route discovery is at least theoretically possible. If it fails, then it won't be because the nodes are not connected, but due to failure of the protocol.

We consider three classes of topologies. The first class are static topologies. Given the constraints that node A, B and C are connected, and that there are at most 5 nodes, this gives 444 topologies, after topologies that are identical up to symmetries are removed. The second class considers pairs of topologies from the first class, in which the second topology can be obtained by adding a new link. This models a dynamic topology in which a link is added. There are 1978 such pairs. The third class considers the same pairs, but now moves from the second topology to the first. This models a link break. Note that after deletion, nodes A, B and C are still connected. In our UPPAAL model a change of topology is modelled by another automaton. It may add or remove a link exactly once, non-deterministically, after the first route request arrives at the destination.

4.2 Properties

This paper considers three desirable properties of any routing protocol such as AODV. The first property is that once all routing messages have been processed a route from the originator to the destination has been found. In UPPAAL syntax this safety property can be expressed as:

```
A[]((tester.final && emptybuffers()) imply
    (node(OIP).rt[DIP].nhop!=0))
```
(1)

The CTL formula `A[]`ϕ is satisfied if ϕ holds on all states along all paths. The variable `node(OIP).rt` models the routing table of the originator node `OIP`, and the field `node(OIP).rt[DIP].nhop` represents the next hop for destination `DIP`. All initiated requests will have been made, iff automaton `tester` is in location `final`, the message buffers are empty iff function `emptybuffers` returns *true*, and the originator `OIP` has a route to node `DIP` iff `node(OIP).rt[DIP].nhop!=0`.

The second property is related, namely that once all messages are processed, then no sub-optimal route has been found. Here, sub-optimal means that the

number of hops is greater than the shortest path. In case that the topology changes, we take the greater distance. In UPPAAL this can be expressed as

```
A[]((tester.final && emptybuffers()) imply
    (node(OIP).rt[DIP].hops<=distance[OIP][DIP]))
```
(2)

Here, the array `distance` encodes the distance matrix. Note, that this fails if the route at the end is sub-optimal. It does not fail if at the end, either an optimal, or no route has been found. If the first two properties are satisfied, it means that it is guaranteed that an optimal route will be found when all messages have been processed. Note that it is known that AODV does not guarantee that optimal routes will be found. Nevertheless, an implementation or modification of AODV can be said to perform better if this property fails for fewer topologies.

The third property is even stronger than the second, namely that no sub-optimal routes will be found at all. It does not hold if a better optimal route replaces a sub-optimal route that was found first.

```
A[](node(OIP).rt[DIP].hops<=distance[OIP][DIP])
```
(3)

If the third property holds, then the second must hold as well. In the experiments we will check all three properties for both originator-destination pairs at once.

4.3 Modifications

The basic UPPAAL model is based on the process algebraic AWN model, which reflects a common interpretation of the RFC with all ambiguities resolved. It is known that AODV does not guarantee that optimal routes will be found, or even any routes at all [5,9].[2] Our experiments quantify how many topologies are affected by these problems, and also what impact slight modifications of the protocol have. We will refer to the basic model as *model 1*, and discuss three proposed variants of AODV.

Forwarding all route replies. It is a known problem that nodes drop route reply messages under certain conditions.[3] During our experiments we found this problem even in the smallest topology, a static linear topology with only three nodes, and only two links: node A is connected to node B and B to node C. Both node B and C initiate a route request to A. For this topology and scenario, UPPAAL finds a counterexample for Property (1), i.e., it is possible that no route will been found when all messages have been processed.

Fig. 3 depicts a message sequence chart of the relevant part of the counterexample. Initially, both B and C initiate a route request for A. We refer to the first request as BA-request, and to the second as CA-request. First, node B sends the BA-request to A and C (Step 1 in Fig. 3), then node C its CA-request to B (Step

[2] AODV proposes to repeat the route discovery process if the first discovery process fails. However, this solution does not solve the problems entirely (see [4]).

[3] This problem has already been raised on the MANET mailing list in Oct 2004 (`http://www.ietf.org/mail-archive/web/manet/current/msg05702.html`).

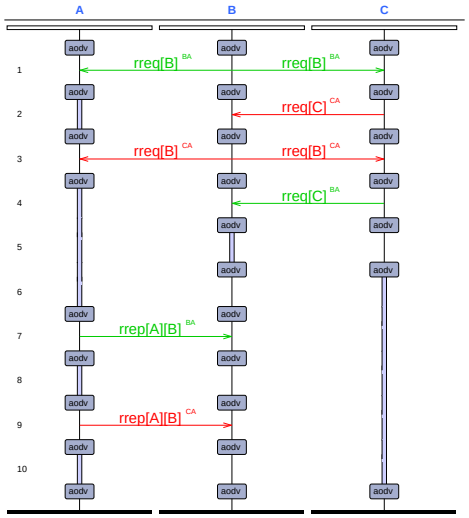

Fig. 3. Message sequence chart illustrating failed route discovery. Wide vertical lines mean that local states do not change in this transition. The superscripts indicate the corresponding originator and destination of the route discovery process.

2). Node B forwards the CA-request (Step 3), node C the BA-request (Step 4). Node C will correctly ignore the CA-request that it received from B, since it is the originator (Step 5). Similarly, B will ignore the BA-request (Step 6). Node A will then reply to the BA-request (Step 7), and node B will update its routing table (Step 8) to include a route to A. Node A will also reply to the CA-request (Step 9), but B will ignore this message (Step 10), since it does not contain new information for B. Node A's reply to the CA-request will not arrive at C.

The discarding of the RREP message happens according to the RFC specification of AODV [11]. It states that an intermediate node only forwards the RREP message if it is not the originator node *and* it uses the RREP to update its route entry to the destination. In this case, node B is not the originator, but it also did not use the route reply to update its route. It already had an optimal route, as a result of the BA-request. This type of problem can arise whenever one node has to relay multiple route requests for the same destination.

A possible solution would be to forward every reply received by a node. Our *model 2* implements this change. Obviously, this increases the number of control messages generated during route discovery. However, this is compensated by the reduced need to repeat sending the route request in case no route has been found, the solution proposed by AODV.[4] In the experiment section we will see that this modification effectively addresses the problem.

Replying to improving requests. Counterexamples found by UPPAAL show that a source for sub-optimal routes is the property of AODV to only reply to the first route request. All subsequent requests with the same request ID (`rreqid`)

[4] Moreover, a repeated route request need not be any more successful than the first.

will be ignored (Line 3 of Tables 1 and 2), even if the subsequent requests arrived via a shorter route. *Model 3* modifies the rule for the handling of route requests. It will not only reply to the first request, but also to a subsequent request (with the same request ID) with an improved hop count.

Recovering from failed replies. Analysis of UPPAAL's counterexamples show that a main reason for failed route discovery is that a node marks a request as having been replied to, even if the node detected the reply failed due to the link being broken in the time between the received request and the sent reply. The node will ignore other requests with the same request ID that may arrive later. *Model 4* introduces two changes: it does not mark a request as seen if the reply fails, and it replies to other requests in the same route discovery process.

This change should be considered with care, since it changes the rules with respect to sequence numbers. These numbers are an essential part of AODV being loop free, and there is currently no guarantee that this change will not violate some essential invariants of the proof [4]. We included the results nevertheless, as they show that there is still significant potential to improve AODV.

4.4 Experimental Results

The experimental results tell for how many topologies UPPAAL could show the absence of counterexamples, and thus allow quantification of the impact of improvements. However, the analysis uses a non-deterministic model, rather than a probabilistic model. For each topology it is reported whether a counterexample exists, but not how likely it is to occur. Neither can we assume that the topologies themselves are randomly distributed. Depending on the application only certain types of topologies might occur in practice. Nevertheless, it is fair to assume that a modification that leads to fewer topologies with counterexamples constitutes an improvement w.r.t. the considered property.

Table 3 presents the results of the experiments. Most relevant for all classes of topologies are Property (1), a route is found, Property (2), no sub-optimal route is found in the end, and the combination of these, i.e., an optimal route is found.

The results demonstrate that the problem of ignoring route replies as described in Figure 3 occurs even for about 50% of all static topologies. *Model 1* satisfies Property (2) only for half of all static topologies. The proposed modification solves this problem entirely for static topologies. The other modifications further improve the quality of the routes; in 99.1% of static topologies Property (2) holds, i.e., the route was in the end always optimal. The slight drop in Property (3) is explained by the fact that in a few cases, where no route was found at all for *model 1*, a sub-optimal route was found in the other models.

The results for static topologies are roughly repeated if we consider topologies in which a link is added. There were a few surprising instances though, in which adding a link was instrumental in finding a sub-optimal route.

The results are, as expected, not quite as positive if a link gets removed. For the baseline model it is only guaranteed for one quarter of all topologies that a route will be found. Relaying all route replies, and not marking requests if the

		Property (1)	Property (2)	Property (3)	Property (1) & (2)	all properties
static	model 1	52.7%	93.2%	50.7%	50.0%	13.5%
	model 2	100.0%	93.2%	47.5%	93.2%	47.5%
	model 3	100.0%	99.1%	47.5%	99.1%	47.5%
	model 4	100.0%	99.1%	47.5%	99.1%	47.5%

		Property (1)	Property (2)	Property (3)	Property (1) & (2)	all properties
add link	model 1	57.5%	90.8%	49.1%	53.3%	18.1%
	model 2	100.0%	90.6%	46.2%	90.6%	46.2%
	model 3	100.0%	97.8%	46.2%	97.8%	46.2%
	model 4	100.0%	96.3%	46.2%	96.3%	46.2%

		Property (1)	Property (2)	Property (3)	Property (1) & (2)	all properties
remove link	model 1	26.7%	90.5%	59.7%	26.2%	6.0%
	model 2	53.0%	89.4%	57.1%	51.2%	28.9%
	model 3	53.0%	93.1%	57.1%	52.8%	28.9%
	model 4	75.4%	94.0%	54.0%	73.8%	41.0%

Table 3. Model checking result for the four models and three classes of topologies. It gives the percentage of topologies for which there exists no counterexample.

reply fails, improves this result. For three quarters of all topologies in which a link was removed it was shown that an optimal route will be found.

The main reason of the failures that remain is that a route reply might get lost because of some intermediate link break on the path back to the destination. A possible solution to this problem could be to maintain a set of back-up routes, or to implement different error responses. However, this requires a significant change and fundamentally changes the characteristics of AODV.

For the experiments we used an Intel Core2 CPU 2.13GHz processor with 2GB internal memory, running Ubuntu 11.04. We used UPPAAL 4.0.13. Of each of the models described in this section, we checked 17600 instances, altogether 70400 instances. As indication of the state space and runtimes, we checked an invariant on all instances of *model 4* for a topology in which a link is removed. These instances have larger state spaces than others, since these scenario have also to trigger the transitions for error handling. The models have an average of 9400 states, the largest model has 475000 states, and the median is 2700. Exploring these state spaces took on average 1.73 seconds user time, at most 81 seconds, and the median was 0.57. These run times show that an automated, systematic and rigorous analysis of reasonable rich routing protocols is feasible.

5 Related Work

Other researchers have used formal specification and analysis techniques to investigate the correctness and performance of AODV; we survey the sample related to model checking.

Bhargavan et al. [2] were amongst the first to use model checking on a draft of AODV, demonstrating the feasibility and value of automated verification of routing protocols. Their investigations using the SPIN model checker revealed that in some circumstances routes containing loops can be created. The proposed variation which guarantees loop freedom were not included in the current standard.

Musuvathi et al. [10] introduced the CMC model checker primarily to search for coding errors in implementations of protocols written in C. They use AODV as an example and, as well as discovering a number of errors, they also found a problem with the specification itself which has since been corrected.

Chiyangwa and Kwiatkowska [3] use the timing features of UPPAAL to study the relationship between the timing parameters and the performance of route discovery. They established a dependence between the lifetime of a route and the size of the network, although their study only considered only the initiate of a single route discovery process, and a static linear topology. In [5], we confirmed some of the problems they discovered, and show their independence of time.

Other researchers have used model checking to analyse other routing protocols. Wibling et al. [13] for example used SPIN and UPPAAL to verify aspects of the LUNAR protocol, which is also used in ad hoc routing for wireless networks. In particular the timing feature of UPPAAL was used to check upper and lower bounds on route finding and packet delivery times. The scenarios considered included a limited number of topology changes where problems were suspected.

De Renesse and Aghvami [12] used SPIN to study the WARP protocol. To reduce the overhead on model checking, various simplifications were imposed on a five-node network, including a single source and destination and limitations on the degree that the network can change.

Fehnker et al. have used the model checker UPPAAL to analyse a TDMA time synchronisation protocol [6]. Similarly to our approach they considered all topologies in a certain class, but did not cover dynamic topologies.

Our approach is in line with these related works. However, it is unique in the sense that our UPPAAL model complements our process-algebraic specification of AODV. As mentioned before, these two approaches to formal protocol modelling, specification and evaluation, if used together, can provide a powerful tool for the development and rigorous evaluation of new protocols and variations, and improvements of existing ones. Currently, our UPPAAL model is derived by hand directly from the AWN specification, but an automatic translation from AWN in the style of Musuvathi et al. [10] is possible, and remains as future work.

6 Conclusions and Outlook

The aim of this ongoing work is to complement by model checking a process algebraic description of WMN routing protocols in general, and AODV in particular. The used description of AODV described in [4] is amongst the first detailed formal models. Having the ability of automatically deriving an UPPAAL model from an AWN specification and thus model checking formal specifications allows the confirmation and detailed diagnostics of suspected errors. The availability of an executable model becomes especially useful in the evaluation of proposed improvements to AODV, as we have shown.

We have sketched possible modifications of AODV, which have been evaluated by formal and rigorous analysis by means of model checking. An analysis of these modifications by means of process algebra is part of future work. We

have set up an environment where we can test a whole bunch of different topologies in a systematic manner. This will allow us to do a fast comparison between standard AODV and proposed variations in contexts known to be problematic.

References

1. 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, LNCS, vol. 3185, pp. 200–236. Springer (2004), http://dx.doi.org/10.1007/978-3-540-30080-9_7
2. Bhargavan, K., Obradovic, D., Gunter, C.A.: Formal verification of standards for distance vector routing protocols. J. ACM 49(4), 538–576 (2002), http://dx.doi.org/10.1145/581771.581775
3. Chiyangwa, S., Kwiatkowska, M.: A timing analysis of AODV. In: Formal Methods for Open Object-based Distributed Systems (FMOODS'05). LNCS, vol. 3535, pp. 306–321. Springer (2005), http://dx.doi.org/10.1007/11494881_20
4. Fehnker, A., van Glabbeek, R.J., Höfner, P., McIver, A., Portmann, M., Tan, W.L.: A process algebra for wireless mesh networks used for modelling, verifying and analysing AODV. Tech. rep., NICTA (to appear), draft available at http://www.nicta.com.au/__data/assets/pdf_file/0006/29292/MeshTR2011.pdf
5. Fehnker, A., van Glabbeek, R.J., Höfner, P., McIver, A.K., Portmann, M., Tan, W.L.: Modelling and analysis of AODV in UPPAAL. In: 1st International Workshop on Rigorous Protocol Engineering (2011), (in press)
6. Fehnker, A., van Hoesel, L., A, M.: Modelling and verification of the lmac protocol for wireless sensor networks. In: Integrated Formal Methods, IFM 2007. LNCS, vol. 4591. Springer (2007), http://dx.doi.org/10.1007/978-3-540-73210-5_14
7. Hiertz, G.R., Denteneer, D., Max, S., Taori, R., Cardona, J., Berlemann, L., Walke, B.: IEEE 802.11s: the WLAN mesh standard. IEEE Wireless Communications 17(1), 104–111 (2010), http://dx.doi.org/10.1109/MWC.2010.5416357
8. Larsen, K.G., Pettersson, P., Wang Yi: UPPAAL in a nutshell. International Journal of Software Tools for Technology Transfer 1(1-2), 134–152 (1997), http://dx.doi.org/10.1007/s100090050010
9. Miskovic, S., Knightly, E.W.: Routing primitives for wireless mesh networks: Design, analysis and experiments. In: IEEE INFOCOM. pp. 2793–2801 (2010), http://dx.doi.org/10.1109/INFCOM.2010.5462111
10. Musuvathi, M., Park, D.Y.W., Chou, A., Engler, D.R., Dill, D.L.: CMC: a pragmatic approach to model checking real code. In: Operating Systems Design and Implementation (OSDI'02) (2002), http://www.usenix.org/events/osdi02/tech/musuvathi.html
11. Perkins, C., Belding-Royer, E., Das, S.: Ad hoc on-demand distance vector (AODV) routing. RFC 3561 (2003), http://www.ietf.org/rfc/rfc3561.txt
12. de Renesse, R., Aghvami, A.: Formal verification of ad hoc routing protocols using SPIN model checker. In: Proceedings of IEEE MELECON'04. pp. 1177 – 1182. IEEE (2004), http://dx.doi.org/10.1109/MELCON.2004.1348275
13. Wibling, O., Parrow, J., Pears, A.N.: Automatized verification of ad hoc routing protocols. In: Formal Techniques for Networked and Distributed Systems – FORTE. LNCS, vol. 3235, pp. 343–358. Springer (2004), http://dx.doi.org/10.1007/978-3-540-30232-2_22