Modelling Ad hoc On-demand Distance Vector (AODV) Protocol with Timed Automata

Sibusisiwe Chiyangwa and Marta Kwiatkowska

School of Computer Science, University of Birmingham,
Birmingham B15 2TT, United Kingdom
{syc,mzk}@cs.bham.ac.uk

Abstract Model checking algorithms for the verification of communication protocols proceed by a systematic and exhaustive state space search. In general, for a concurrent system, the size of the state space grows exponentially with the number of components in the system. Thus, naive verification algorithms, which simply traverse the complete state space using an explicit state representation, do not scale up to industrially relevant problems. In this paper, we aim to arrive at a methodology for verifying real-world concurrent communication protocols focusing on their data, control and real-time aspects. We describe how UPPAAL can be used to verify some of the timing properties of Ad hoc On-demand Distance Vector protocol (AODV).

Keywords: automatic verification, model checking, real-time, timed automata, UPPAAL.

1 Introduction

Communication services are becoming indispensable and are penetrating all fields of life (tele banking, chip cards, electronic mail, control of industrial facilities, traffic control systems, etc). It is important that systems work correctly, reliably and securely.

The primary goal of this work is to formulate a methodology for the verification of real-life communication protocols using *Ad hoc On-demand Distance Vector* AODV [DPR02] as our test protocol. AODV is one of *mobile ad hoc network* (MANET) protocols. A mobile ad hoc network is a collection of nodes, where each node communicates over wireless channels and is capable of movement. Nodes participating in such an “ad hoc” network communicate on a peer-to-peer basis. In our case, “ad hoc” means unplanned and/or established for one specific reason or occasion. The nodes are free to move randomly and organise themselves arbitrarily. An ad hoc network is formed without a-priori knowledge or planning. Nodes belonging to an ad hoc network do not share any central authority or support structure. Ad hoc networks can be set up anywhere and anytime. Basically, the network is a collection of participating mobile nodes in a wireless environment forming a temporary network without need for any infrastructure.

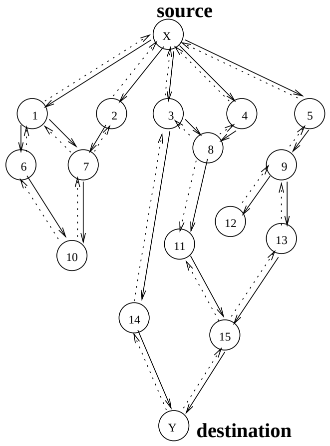
Most of the work that has been done so far concerning AODV has focused on the data aspects of the protocol. [BGO99] summarised some of the routing loops in earlier versions of AODV. Murphi was used to detect some of these bugs and Spin to confirm the bugs. Some bugs were discovered by manual inspection (Joshua Broch and Dave Maltz, CMU), then Spin and Murphi were used to confirm them. [MPC+02] found one routing loop in AODV protocol specification and two routing loops in one of the AODV implementations using CMC. [BGO99] produced a rigorous proof using a theorem prover HOL and Spin to verify AODV. To complement on the data aspects of the protocol, we would like to look at the timing properties of AODV.

Ad hoc On-demand Distance Vector Protocol in brief:
AODV [DPR02] is an *on-demand* or *reactive* (on-the-fly formation of networks) protocol designed for use in ad hoc mobile networks. It allows users to find and maintain routes to other users in the network whenever such routes are needed [PR99]. AODV is an ad hoc IP routing protocol that supports unicast, broadcast, and multicast. The routing decisions are made using distance vectors. In *distance vector* routing, each router contains distances, measured in hops, to all available routers in its routing table. Since this in an on-demand distance vector, routers only maintain distances of only those destinations that they need to contact or relay information to.

AODV uses *sequence numbers*, which serve as time stamps. They allow nodes to compare the "freshness" of their routes. Every time a node originates a *ROUTE REQUEST*, (RREQ) or a *ROUTE REPLY* (RREP), it increments its own sequence number. Each node records all sequence numbers of its active neighbours. A higher sequence number signifies a fresher route. As a reactive protocol, AODV uses *route discovery* and *route maintenance.*

During route discovery, when a node requires a path to destination, it broadcasts a RREQ message to its neighbours which includes latest known sequence number for that destination. Each node receiving the message creates a reverse route to the source node. RREQ propagation is illustrated in Figure 1.

The destination node or an intermediate node that has an up-to-date route sends back ROUTE REPLY message which includes the number of hops in between and the destination's sequence number. Each node receiving the route reply message creates a forward route to the destination. Thus, each node remembers only the next hop required to reach any of the hosts, not the whole route. RREP propagation is illustrated in Figure 2.

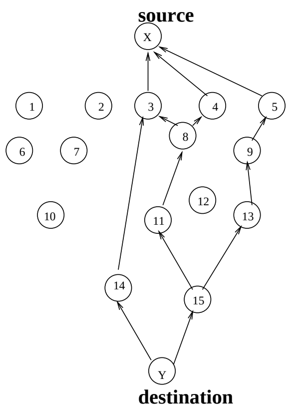


Source node X broadcasts to its neighbours nodes 1, 2, 3 ,4 and 5. Node 1 later broad casts to its neighbours nodes X, 6 and 7. This goes on until we get to nodes 14 and 15 which will both sent the RREQ to the destination node Y.

Dotted backwards arrows indicate that a node is sending a RREQ to a node that sent it that RREQ, since it is also its neighbour. These RREQ are ignored by the initial forwarding nodes as they are treated as duplications.

RREQ propagation

Fig. 1. AODV Route Discovery (RREQ propagation)



RREP propagation

The backward arrows represent the RREP being sent back to the source node X. Because nodes 10 and 12 have no routes to Y, nodes 1, 2, 6, 7 and 12 are left out.

Route	Hop Count
X→3 →14→Y	3
X→3→ 8→10→15→ Y	5
X→4→ 8→10→15→ Y	5
X→5→ 9→13→15→ Y	5

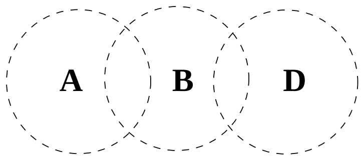
Route **X→3 →14→Y** is chosen since it has the least hop count. We assume the destination sequence number is the same for all the routes.

Fig. 2. AODV Route Discovery (RREP propagation)

During route maintenance, AODV broadcasts periodic RREPs with TTL=1 called *HELLO* messages for local connectivity. Failure to receive HELLO messages from an active neighbour within a pre-defined amount of time indicates the loss of a link to that neighbour. Whenever this is detected, the source is notified with a ROUTE ERROR (RERR) message which includes infinite metric for the destination and an incremented destination sequence number. In that case, the source node initiates a new route discovery. This protocol is loop free, and link breakages result in notification of demanding nodes immediately. AODV avoids the Bellman-Ford *Counting-to-infinity* problem.

2 Modelling AODV

We consider a topology with 3 nodes, A, B and D, where A is the source node and D is the destination node. B is the intermediate node. Below we illustrate the topology we will work with.



Node	Neighbours
A	B
B	A, D
D	B

Fig. 3. 3-node topology

The source, A, may broadcast a RREQ, and in the model only B gets the RREQ. Since B is not the intended destination, it will rebroadcast it again and both A and D will get this RREQ. A will ignore the RREQ. Since D is the intended destination, it will reply with a RREP. This RREP is unicasted to B who will later unicast it to A. If, at any point, B has a fresh enough route to D, it may reply to the RREQ. These RREQ and RREP packets cannot be lost at any time within the network in a perfect model. In a lossy model, packets may be lost, and in such a case the sender A will time out and re-send the RREQ up to RREQ_RETRIES, after which the sender can conclude that a route to the desired destination does not exist.

Assumptions about our model:
To formulate a feasible model we make some assumptions about our underlying communication mechanisms.

1. Node A is the only source node and node D is the only destination node.
2. At any time there is at most one packet (RREQ, RREP or RERR) in the network.
3. Time does not elapse when a node is processing packets (processing time is zero).
4. Packets may not be lost if a link exists between two nodes in a perfect model.

We model the above topology using timed automata [AD94], a formal notion to model the behaviour of real-time systems. Timed automata are state-transition diagrams with a set of edges between locations denoting transitions, annotated with timing constraints using finitely many real-valued clock variables. The clocks may be considered as the system clocks of a concurrent system. They are assumed to proceed at the same rate and measure the amount of time that has elapsed since they were reset. The clock values can be tested (compared with natural numbers) and reset (assigned to 0).

We then verify the model using the UPPAAL [LPY97] model checker, *Uppaal2k* [PL00]. We intend to use data abstraction to reduce the size of our model. Each node has an unbounded *sequence number* and *broadcast id* to uniquely identify each RREQ. Since we have assumed that we will have at most one packet in the network at any given time, we will drop the *broadcast id* from the model. As long as there is a valid route between two endpoints, AODV does not play any role [DPR02] and hence our model has no actual data packets. To do an effective verification with infinite variables we need to bound our search, thus the *sequence number* will be bounded.

We define the lifetime values of valid routes as integer variables. Defining them as individual clocks would greatly increase the size of the model. These lifetimes need to be compared to UPPAAL clocks for timeouts. Clocks in UPPAAL cannot be compared to integer variables [LPY97]. Therefore, we substitute our global clock with an integer clock [Zou00]. The value of this integer clock increases by one time unit with every tick of the UPPAAL clock. This integer clock cannot be used in invariants because invariants may only contain Uppaal2k clocks. The integer clock is another unbounded counter that needs bounding as well. We stop simulation and verification when the integer clock exceeds the predefined limit. This is to avoid problems with roll-over values. We have also scaled down the timing constants.

The UPPAAL model consists of an integer *clock*, a generic *node*, and *link_failure* templates. We make use of a template node because in AODV all the nodes can be source nodes, routers or destination nodes, and thus share similarities. Three instances of this template node will define our 3-node topology.

3 The UPPAAL Model

Integer clock template

In Figure 4 above, a node will look at the integer clock before processing packets. The channel *b* enables the integer clock to synchronise with the nodes. With every tick the integer clock's value *t* is incremented by one. When the integer value reaches *range1* it stops. We simulate and verify up to *range1*, not *range* (our predefined maximum), because the RREQ timer is at most *NET_TRAVERSAL_TIME* more than the current time. We only go up to *range1* which is equal to *range* - *NET_TRAVERSAL_TIME* to avoid roll-over problems.

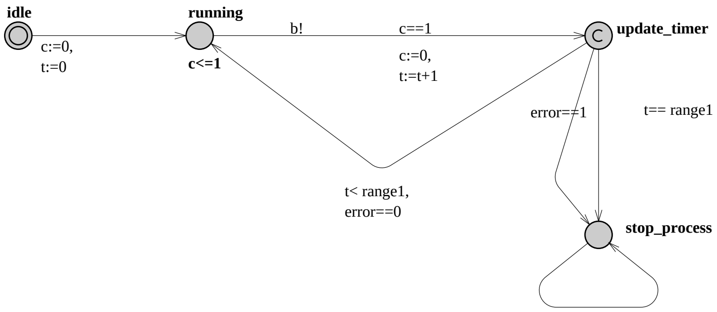


Fig. 4. Integer clock Template

If there is a link break, the source node is notified of this link break and will try to re-discover an alternative route to the destination. Our topology has only one route to the destination node. No route will be discovered after this link break. Our integer clock allows us to stop when the source node has tried *RREQ_RETRIES* times without success. Thus the node can conclude that no route exists after a link break.

Node template

Each node has its own local variables, for example a routing table. UPPAAL does not allow us to model 2-dimensional arrays (arrays of arrays). Our routing table is a 2-dimensional array. Since we could not model it as one data structure, we had to model its columns as separate one-dimensional arrays as depicted by the diagram below.

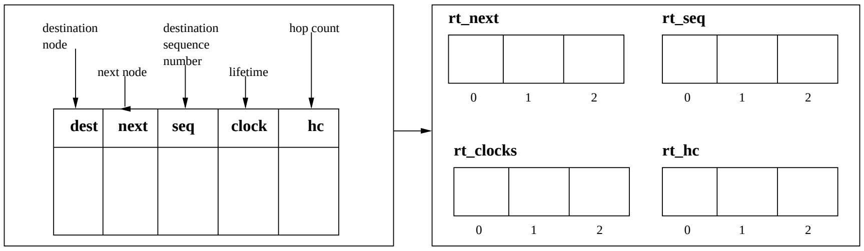


Fig. 5. The Routing Table

rt_seq is the routing table's destination sequence column, *rt_clocks*is the routing table's lifetime column, *rt_hc* is the routing table's hop count column and *rt_next* is the routing table's next node column. Since we have three nodes in the system, each node has three cells in each of its routing table's arrays. A node will have a route to itself but its hop count in the *rt_hc* array will always be zero. In our model this means no route to that destination. Each node keeps track of its own sequential number and the number of RREQ attempts.

A node will perform the following tasks:
- send a RREQ messages
- receive and process a RREQ messages
- send a RREP messages
- receive and process a RREP messages
- send RERR messages
- receive and process RERR messages
- send HELLO messages
- receive and process HELLO messages
- manage its own routing table

The loss of a messages is modelled by an internal transition (unlabelled transition in UPPAAL), since the loss itself does not constitute an observable event. The full node template is in Appendix A.

link_failure template
In our model, the source node looks for a route to a destination node and when the route is available, waits for it to expire, then re-discover it again. Basically we have a *discover route - wait to expire - re-discover route* cycle. After a number of these cycles we break the link between node B and D. The *link_failure* template synchronises with node B to break the link, and then the route error process is initiated.

4 Timing properties

We extract our properties from the AODV draft [DPR02]. The draft states that AODV allows mobile nodes to respond to link breakages and changes in the network topology in a timely manner. Routes are also found in a timely manner. We formally express some of the properties below and use our model (perfect and lossy models) to verify the properties.

1. Establish absence of deadlock (this requires exploration of the full state space)

   ```
   A[] not deadlock  (safety property)
   ```

2. It is possible to find routes in a timely manner

   ```
   E<> (node_0_1.got_route ) (reachability property)
   ```

   Is it possible to receive a RREP (node_0_1.got_route) when the RREQ timer has not timed out (rreq_timer <= t)?

3. No routes are determined when we have an active route in the routing table. We add a *catch* state, so that we can test for the reachability of that state. When we are in the state *want_route*, we check the routing table to see if we have a valid route to the desired destination. If we have, we then move to our catch *state*. We can thus test to see if the system cannot reach the *catch* state. If the property is satisfied, then we can conclude that the system will not look for a route to a destination whilst it has a valid route in its routing table.

   ```
   A[] not (node_0_1.caught) (reachability property)
   ```

4. There is no route to a destination after a link break on the one and only route to the destination node.

   ```
   A[] not (node_0-1.got_route and cycle >= break)
   ```

   When there is a link break on an active route, the source node in our topology should then never find a route to that destination. Thus, we test to see if there is no route to the destination after a link break. This link break is between nodes B and D. If we change our destination to node B, then a route to node B should be found. We modify property 2 to get

   ```
   E<> (node_0_1.got_route and cycle >= break)
   ```

   which is saying that we can get a route after the link break. The dual of this is the required property above. This follows from the rule $\neg E <> \phi \equiv A[]\neg\phi$.

Analysis of the results
Using a machine (Preston) with 4GB of memory, we verified the current corrected version of AODV. With a perfect model (loss of messages not allowed), we could verify up to a maximum range of 100 000. With a lossy model we could only verify up to a maximum of 225 (which corresponds to breaking the link after 25 cycles) before we ran out of memory. We did not find any bugs in the current version of AODV for this configuration.

Work in progress
We plan to work with earlier versions of AODV and see if we detect known and/or new errors . Our model currently has only three nodes. We also intend to extend the number of nodes in the model and experiment with different topologies. To make our model more tractable, we intend to find ways of abstracting some of the data in the model to reduce the model size.

5 Conclusion

We have been successful with finding a tractable timed automata model of AODV. In AODV, sequence numbers and clock values are unbounded. Bounding these variables reduces the size of the model and prevents deadlock due to roll-over problems. With an unbounded model (sequence numbers and clock values unbounded), we could not establish the absence of a deadlock. We also abstracted the broadcast id from the model. Because we consider one packet at a time in our model, broadcast id does not play any role.

We have verified some of the timing properties of the current version of AODV using UPPAAL. Interestingly, we could verify that if we break the link between node B and node D, there would be no route to D thereafter. If we changed our destination node to B a route could be found. Our challenge now is to add more nodes to the network so that we not only have one route to the destination node, but several routes. We would then want to verify that if all the links were active, the optimal route would be chosen (i.e route with lower hop count). Breaking this link should then enable the source node to use one of the remaining links.

UPPAAL uses very restricted data structures, forcing us to find alternative ways of expressing some of our complex data structures. This resulted in an increase in model size. With the present version of Uppaal, we need to find good abstraction techniques to reduce the model size if we are to consider larger configurations.

References

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[BGO99] Karthikeyan Bhargavan, Carl A. Gunter, and Davor Obradovic. Formal Verification of Standards for Distance Vector Routing Protocols. Technical report, Department of Computer and Infromation Science, University of Pennsylvania, December 1999. Working Paper. Presented in the Recent Research Session at Sigcomm 1999.

[DPR02] Samir R. Das, Charles E. Perkins, and Elizabeth M. Royer. Ad hoc On-Demand Distance Vector (AODV) routing, January 2002. INTERNET DRAFT draft-ietf-manet-aodv-12.txt.

[HR00] Michael R.A. Huth and Mark D. Ryan. *Logic in Computer Science. Modelling and reasoning about systems.* Cambridge University Press, 2000.

[LPY97] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[MPC+02] Madanlal Musuvathi, David Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, December 2002.

[PL00] Paul Pettersson and Kim G. Larsen. Uppaal2k. *Bulletin of the European Association for Theoretical Computer Science*, 70:40–44, February 2000. http://www.uppaal.com.

[PR99] Charles E. Perkins and Elizabeth M. Royer. An Implementation Study of the AODV Routing Protocol. In *Proceedings of the IEEE Wireless Communications and Networking Conference, Chicago, IL*, September 1999.

[Zou00] Bohumir Zoubek. Fault-Tolerant Real-Time Protocols. Master's thesis, Schools of Computer Science, University of Birmingham, 2000.

A Node Template

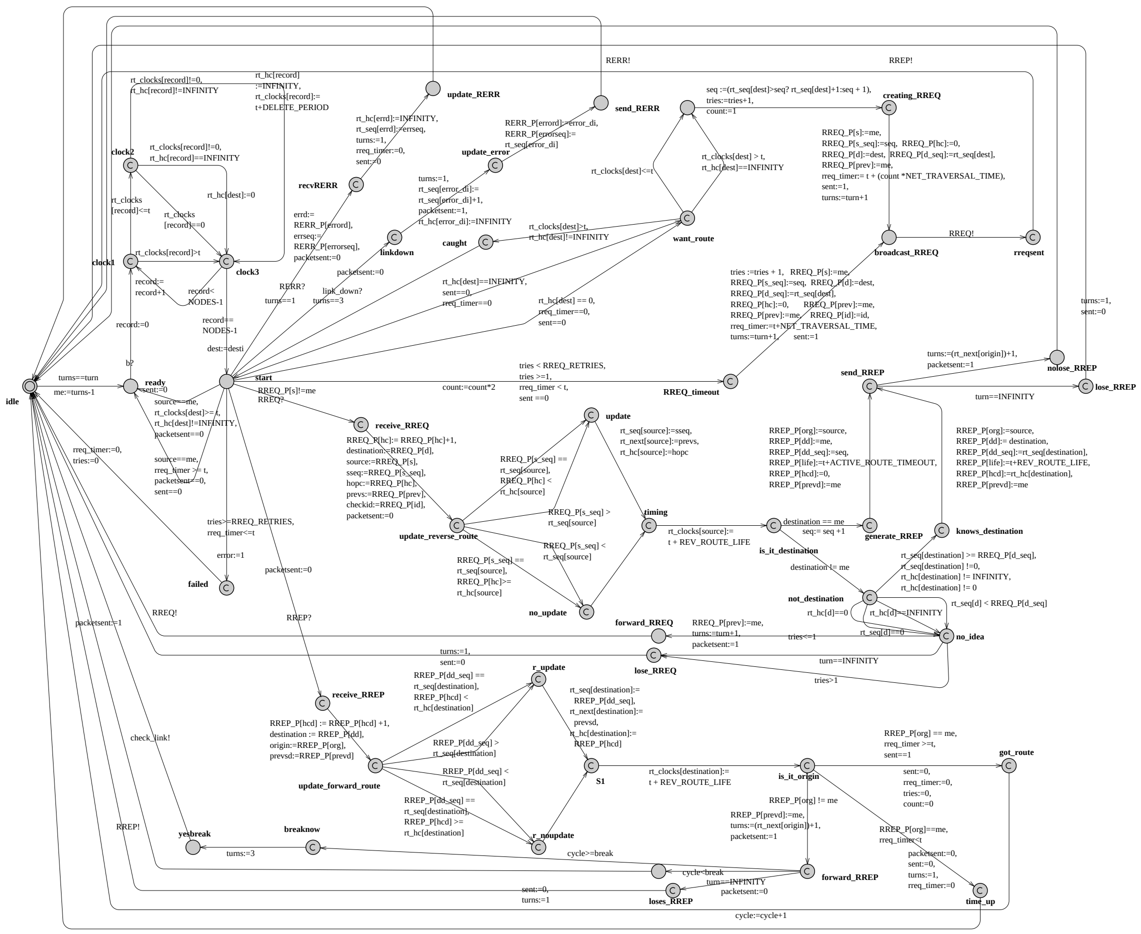


Fig. 6. Node Template