International Journal of Networking and Computing – `www.ijnc.org`
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 7, Number 1, pages 29–49, January 2017

State space reduction techniques for model checking of MANET protocols

Hideharu Kojima, Yuta Nagashima, Tatsuhiro Tsuchiya
Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita, 565-0871, Japan

Received: July 25, 2016
Revised: November 28, 2016
Accepted: December 21, 2016
Communicated by Akihiro Fujiwara

Abstract

A Mobile Ad hoc Network (MANET) is a network that consists of mobile nodes and is autonomously managed without infrastructure base stations such as access points. MANETs have started being used as part of safety critical applications. A Vehicular Ad hoc Network (VANET) used in automated driving systems is such an example. In such applications, defects in the network protocol may cause serious social problems. Model checking, a state search-based verification technique, has proven to be effective in finding faults in complex system designs, such as communication protocols. However it is challenging to apply this technique to MANET protocols, because a MANET can have a number of different network topologies, thus resulting in the state explosion problem very easily. In this paper we propose a modeling technique to mitigate this problem using the AODV protocol as a running example. MANET protocols, such as AODV, typically enforce a source node that wishes to establish a route to the destination to retry the route establishing process some fixed number of times in face of failures. We show that to model check the protocol's behavior in these retries it suffices to consider only the last trial. The results of experiments using the SPIN model checker show that using the proposed technique significantly reduced the time and memory usage compared to standard full state exploration and allowed us to model check the protocol with up to five nodes.

Keywords: MANETs, model checking, state space reduction;

1 Introduction

A *Mobile ad hoc network (MANET)* is a network that consists of mobile nodes. A collection of these mobile nodes autonomously manages the MANET without infrastructure base stations, such as access points. These nodes can move and go away from the communication range of each other; thus the topology of a MANET varies dynamically. Examples of MANETs include vehicular ad hoc networks (VANETs). In a VANET an automobile serves as a node. Automated driving systems are an example of an application of VANETs. As MANETs serve as social infrastructures, their failures can cause serious social problems. Aiming to prevent failures resulted from design defects of MANET communication protocols, we focus on the verification of these protocols.

Many MANET routing protocols have been developed so far [7][4]. These protocols are classified into three groups according to the type of route maintenance. The first group consists of *reactive*

[0]This paper is an extended version of [28].

protocols including AODV [33] and DSR [26]. In a reactive protocol a communication route is established on demand: The source node, which wishes to send data to another node, tries to establish a route to the destination node before the data transfer. After the source node finishes sending the data, the route will be discarded. The second group consists of proactive protocols, such as OLSR [11] and TBRPF [32], where all nodes keep tracks of routes to other nodes in the network, while the third groups consists of hybrid protocols which have features of both groups. In this work we limit our interest to reactive protocols, since the protocols of this type have received most attention among the three groups.

Model checking is a common method for verifying communication protocols. Mature model checking tools, such as SPIN [25] and UPPAAL [29], are readily available. These tools can mechanically verify whether a given property holds or not by searching all possible states of a target protocol design. This comprehensiveness is especially useful in the verification of MANET protocols, because the topology of a MANET can be various and dynamically change. Unlike model checking, simulations or real device experiments can check only a fraction of the state space.

However, it is challenging applying model checking to MANET protocols as it results in state space explosion very easily. In this paper we propose a modeling technique to mitigate this problem in the model checking of reactive MANET protocols. A reactive protocol enforces a source node which wishes to establish a route to the destination to retry the route establishing process some fixed number of times in face of failures. We show that to model check the protocol's behavior in these retries it suffices to consider only the last trial. This is based on the observations that when initiating the last trial, the state of a node is of either one of a few possible patterns and that these patterns subsume those that occur at the previous trials. To show the usefulness of the proposed technique, we will later show the results of experiments using the SPIN model checker [25].

In this paper we explain our technique for a particular MANET protocol, namely, the AODV protocol, which is one of the best known reactive protocols. Although applying the technique to other protocols is left as future work, we believe that this is not difficult because AODV shares substantial traits with other reactive protocols.

2 Routing Protocol AODV

In this section, we explain AODV on which we focus in this paper. Nodes in a MANET behave as follows. When a source node communicates with another node, the source node needs to establish a route to that destination node. To this end, the source node broadcasts a request packet *rreq*. A node that received *rreq* compares its own ID with the destination ID specified by *rreq*. If the node's ID is the destination ID, then the node will send a reply packet *rrep* to the source node. If not, the node forwards *rreq* to its neighbor nodes by broadcasting it.

2.1 Packets

In addition to data packets, AODV uses three types of control packets, namely *rreq*, *rrep*, and *rrer*. Here we explain *rreq* and *rrep* because they are used in establishing a route from a source node to its destination node. The third kind of packets, *rerr*, is used to, for example, notify that an already established route has become unavailable. As the focus of our paper is on verification of route establishing processes, we do not consider *rerr* hereafter.

rreq: Route REQuest packet

An *rreq* is a six-tuple $rreq = (src, sndr, dest, seq_S, seq_D, hops)$ where:

src : The ID of the source node that is establishing a route to the destination node.

sndr : The ID of the sender node that has sent the packet.

dest : The ID of the destination.

seq$_S$: The sequence number.

seq$_D$: The sequence number stored in an entry of the routing table of the source node if the source node has already established a route. If not, this value is 0.

hops : The number of hops from *src*.

A source node establishes a route to its destination node as follows. First, the source node broadcasts an *rreq* to its neighbors. When a node that is not the destination receives the *rreq*, the node will forward it by broadcast or simply discard it, depending on its routing table. In the case of forwarding, the node modifies *rreq* by replacing the *sndr* element with its own ID and then forwards the new packet.

rrep: Route REPly packet

When the destination node receives *rreq*, it will generate a reply packet *rrep* then send *rrep* to one neighbor by unicast. An *rrep* travels back from the destination node to the source node on the route. An *rrep* is a six-tuple *rrep* = (*dest*, seq_D, *src*, *sndr*, *next*, *hops*). *rrep* is a reply packet for *rreq*. These elements of an *rreq* are determined as follows:

dest : The ID of the destination node (same as *rreq.src*)

seq$_D$: The sequence number (same as $rreq.seq_S$)

src : The ID of the source node (same as *rreq.dest*)

sndr : The ID of the sender node

next : The ID of the node that should receive this packet

hops : The number of hops from src

2.2 Routing Table

A node has one routing table which holds entries. Each entry indicates a pair of a destination node and the next hop node to which the node should forward a packet for the destination node. When a node receives a packet, the node will look for an entry that matches the destination node of the received packet. If such an entry is found, then the node will forward the packet to the next hop node specified by the entry. An entry e is a four-tuple $e = (dest, next, seq, hops)$ where

dest : The ID of the destination node.

next : The ID of the next hop node to which a packet to *dest* is forwarded.

seq : The sequence number.

hops : The hop count from *dest*.

Upon receiving a packet, the node performs either one of the three operations: adding an entry, updating an entry, or discarding the packet. Which of these operations is performed depends on the entries of the routing table and the incoming packet, as described below. We let *rt* denote the routing table and assume that no two entries have the same *dest*, i.e., $\forall e_1, e_2(\neq e_1) \in rt : e_1.dest \neq e_2.dest$. The sequence number *seq* is used to determine whether or not a received packet is newer than its corresponding entry. If the packet's sequence number is greater than that of the entry, then the entry is updated because the packet is newer. In the rest of this subsection, we describe in detail when a new entry is added and when an existing entry is updated. We denote by *id* the ID of the node that has received the packet. In practice, an IP address is used as an ID; but for simplicity of presentation, we often use small integers as concrete examples of IDs. The *hops* entry represents the hop count from the destination node, that is, the number of hops needed to reach the destination.

2.2.1 Adding an entry

The node that received a packet adds a new entry to the routing table if the following condition holds. If the packet received is an *rreq*, then the condition is:

$$\forall e \in rt : e.dest \neq rreq.src \ \wedge \ rreq.dest \neq id \tag{1}$$

or

$$\forall e \in rt : e.dest \neq rreq.src \ \wedge \ rreq.dest = id \tag{2}$$

The node generates a new entry e, which is defined by (3), and adds it to its own routing table when either eq.(1) or eq.(2) is true. Eq.(1) holds when the node is a forwarding node, whereas eq.(2) holds when the node is the destination node.

$$e = (dest, next, seq, hops) = (rreq.src, rreq.sndr, rreq.seq_S, rreq.hops) \tag{3}$$

If the packet received is an *rrep*, then the condition is:

$$\exists e1 \in rt : (e1.dest = rrep.dest \ \wedge \ \forall e2 \in rt \backslash \{e1\} : e2.dest \neq rrep.src) \wedge rrep.next = id \tag{4}$$

The node generates a new entry e, which is defined by (5), and adds e to its own routing table when eq.(4) is true.

$$e = (dest, next, seq, hops) = (rrep.src, rrep.sndr, rrep.seq_D, rrep.hops) \tag{5}$$

2.2.2 Updating an entry

Upon receiving a packet *rreq* or *rrep*, the node updates an entry e such that $e.dest = rreq.src$ or $e.dest = rrep.src$ if the following condition holds.

For an *rreq*:

$$\exists e \in rt : (e.dest = rreq.src \ \wedge \ e.seq < rreq.seq_S) \ \wedge \ rreq.dest \neq id \tag{6}$$

or

$$\exists e \in rt : (e.dest = rreq.src \ \wedge \ e.seq < rreq.seq_S) \ \wedge \ rreq.dest = id \tag{7}$$

The node updates e as defined by (8) when either eq.(6) or eq.(7) is true. Eq.(6) applies when the node is a forwarding node, whereas eq.(7) holds when the node is the destination node.

$$e = (dest, next, seq, hops) = (dest, rreq.sndr, rreq.sep_S, rreq.hops) \tag{8}$$

For an *rrep*:

$$\begin{gathered} \exists e1 \in rt : (e1.dest = rrep.dest \ \wedge \\ \exists e2 \in rt \backslash \{e1\} : (e2.dest = rrep.src \wedge e2.seq < rrep.seq_D)) \ \wedge \ rrep.next = id \end{gathered} \tag{9}$$

When eq.(9) is true, the node updates $e2$ as follows.

$$e2 = (dest, next, seq, hops) = (dest, rrep.sndr, rrep.seq_S, rrep.hops) \tag{10}$$

2.2.3 Discarding a received packet

Even if an entry that corresponds to the received packet exists, the packet is simply discarded if the entry has already been updated. This occurs if none of the conditions shown in 2.2.1 and 2.2.2 holds.

```
    AODV behavior of a node
1:  Constant $X$: set of ID
2:  var $RT$ : array $X$ of Routing Table
3:  var $CH$ : array $X$ of Channel
4:  var $e1$, $e2$ : entries of a Routing Table
5:  var pkt : pkt$\in \{rreq^l_m, rrep^l_m\}(l \in \{1,2,3\}, m \in X)$a received packet
6:  node $k$: $X$
7:  begin
8:  while(true) begin
9:    $CH[k]$?pkt;
10:   $e1$:=exist(pkt.src);$e2$:=exist(pkt.dest);
11:   pkt.type=req$\wedge$pkt.dest$\neq k \wedge e1$=null$\rightarrow$ broadcast(pkt),insert(pkt)
12:   []pkt.type=req$\wedge$pkt.dest$\neq k \wedge e1 \neq$null$\wedge e1$.seq<pkt.seq$\rightarrow$broadcast(pkt),update($e1$,pkt)
13:   []pkt.type=req$\wedge$pkt.dest=$k \wedge e1$=null$\rightarrow$insert(pkt),reply(pkt)
14:   []pkt.type=req$\wedge$pkt.dest=$k \wedge e1 \neq$null$\wedge e1$.seq<pkt.seq$_S$ $\rightarrow$update($e1$,pkt),reply(pkt)
15:   []pkt.type=rep$\wedge$pkt.dest$\neq k\wedge$pkt.next=$k\wedge e1$=null$\wedge e2\neq$null$\rightarrow$unicast($e2$,pkt),insert(pkt)
16:   []pkt.type=rep$\wedge$pkt.dest$\neq k\wedge$pkt.next=$k\wedge e1\neq$null$\wedge e2\neq$null$\wedge e1$.seq<pkt.seq$_D$
        $\rightarrow$unicast(pkt),update($e2$,pkt)
17:   []else$\rightarrow$skip;
18: end while
19: end
```

Figure 1: Program for forwarding node and destination node

2.3 Program Description

The program in Fig. 1 represents the behavior of the destination node and forwarding node in AODV. The program consists of a set of objects and a set of actions. An action may have a guard, in which case it is described as *guard* $\rightarrow$ *statement*. The *guard* is a boolean expression over the states of objects in the program. When the guard is *true*, the *statement* can be executed. If some actions are described as

$$guard1 \rightarrow statement1[]guard2 \rightarrow statement2[]guard3 \rightarrow statement3$$

then, one of the actions whose guard is *true* is executed. If two or more guards become *true* at the same time, one action is nondeterministically selected to execute.

The objects in the program in Fig. 1 include:

- $RT[k]$: the routing table of node $k(\in X)$, which is a set of entries, i.e., $RT[k] = \{e1, e2, \ldots\}$.
- pkt: the packet received through incoming channel $CH[k]$

X is a set of node IDs in the network. $CH[k]$ is the incoming channel for node $k(\in X)$. $CH[k]$?pkt in line 9 represents an action that receives a packet and stores it in pkt when a packet exists in $CH[k]$. The type of pkt is either *rreq* or *rrep*.

The functions used in the program are described below.

- exist(id)($id \in X$): returns an entry e such that $e.dest$=id from $RT[k]$. Returns null if no such entry exists.
- insert(pkt): creates an entry e from pkt and adds it to $RT[k]$. If the packet type of pkt is *rreq*, this function creates an entry e as specified by (3). If pkt is *rrep*, this function creates e as specified by (5).

- update(e,pkt): updates e based on pkt. If pkt is $rreq$, e is updated as defined as (8). If pkt is $rrep$, e is updated as defined as (10).

- broadcast(pkt): replaces pkt.$sndr$ with k and puts pkt into $CH[l]$ for all $l \in X'$ where X' is the set of neighbor nodes that can communicate with k at the time of the broadcast. X' can be any subset of $X \backslash \{k\}$, i.e., $X' \subseteq X \backslash \{k\}$.

- unicast(e,pkt): replaces pkt.$sndr$ with k and pkt.$next$ with $e.next$ and puts pkt into $CH[e.next]$.

- reply(pkt): obtains an entry e by executing exist(pkt.src), generates $rrep = (dest,\ seq_D,\ src,\ sndr,\ next,\ hops) = (e.dest, e.seq, k, k, e.next, 0)$ and puts it into $CH[e.next]$.

As shown in Fig. 1, the program consists of a big while loop which is executed in response to the reception of a new packet. Table 1 explains each action in the while loop.

Table 1: Actions of the program in Fig. 1

Line No.	Guard (Condition in Sec. 2.2)	Statement
9	none	picking up a packet from channel $CH[k]$
10	none	finding entries $e1$ and $e2$ from $RT[k]$ by using exist(id)
11	Eq.(1)	When node k is not the destination node of the received packet pkt, node k adds an entry e generated from the received pkt by using insert(pkt) and broadcasts the new packet.
12	Eq.(6)	When node k is not the destination node of the received packet pkt, node k updates the entry $e1$ by using updates($e1$,pkt) and broadcasts the new packet.
13	Eq.(2)	When node k is the destination node of the received packet pkt, node k adds an entry e generated from the received packet pkt by using insert(pkt) and generates a new $rrep$ and sends it by using reply(pkt).
14	Eq.(7)	When node k is the destination node of the received packet pkt, node k updates the entry $e2$ and sends a packet by unicast(pkt).
15	Eq.(4)	When the type of the received packet is $rrep$, node k generates an entry e and adds it by using insert(pkt). Then, node k sends it by unicast(pkt).
16	Eq.(9)	When the type of the received packet is $rrep$, node k updates the entry $e2$ by using update($e2$,pkt). Then node k sends $rrep$ by unicast(pkt).
17	none	Node k discards the received packet.

2.4 Illustrative Example

We illustrate the route establishment process in AODV using Fig. 2. The example network consists of one source node S, one destination node D and two forwarding nodes 1 and 2. Hence X, the set of the node IDs, is $\{S, D, 1, 2\}$. Fig. 2(a) represents the situation where D receives an *rreq* originally initiated by S. Fig. 2(b) shows that S receives an *rrep* from D.

These tables are initially empty. The sequence numbers seq_S and seq_D in the *rreq* sent by S are 0.

(a)-(1): S broadcasts $rreq = (S, S, D, 0, 0, 0)$, then both node 1 and node 2 receive it. As the routing tables of node 1 and node 2 were empty, a new entry is added to these routing tables. $\{e^1_{(req,S)}\}$ in the third column is a symbolic representation of the routing table which will be explained in Section 4.

Routing tables

RTs	$(dest, next, seq, hops)$	
$RT[S]$		$\emptyset$
$RT[1]$	$(S, S, 0, 1)$	$e^1_{(req,S)}$
$RT[2]$	$(S, S, 0, 1)$	$e^1_{(req,S)}$
$RT[D]$		$\emptyset$

(a)-(2): Node 1 broadcasts an $rreq = (S, 1, D, 0, 0, 1)$ to forward it. On receiving the *rreq* D adds an entry $(S, 1, 0, 2)$ to its own routing table $RT[D]$.

Routing tables

RTs	$(dest, next, seq, hops)$	
$RT[S]$		$\emptyset$
$RT[1]$	$(S, S, 0, 1)$	$e^1_{(req,S)}$
$RT[2]$	$(S, S, 0, 1)$	$e^1_{(req,S)}$
$RT[D]$	$(S, 1, 0, 2)$	$e^1_{(req,1)}$

(a)-(3): Node 2 behaves similarly to node 1. Node 2 broadcasts $rreq = (S, 2, D, 0, 0, 1)$ to forward it. Although node 1 receives the *rreq* from node 2, node 1 discards it.

Routing tables

RTs	$(dest, next, seq, hops)$	
$RT[S]$		$\emptyset$
$RT[1]$	$(S, S, 0, 1)$	$e^1_{(req,S)}$
$RT[2]$	$(S, S, 0, 1)$	$e^1_{(req,S)}$
$RT[D]$	$(S, 1, 0, 2)$	$e^1_{(req,1)}$

(b)-(4): D generates an $rrep = (S, 0, D, D, 1)$ from entry $e = (S, 1, 0, 2)$ and sends it by unicast to node 1 which is the *next* node of e. Receiving the *rrep* from D, node 1 adds entry $(D, D, 0, 1)$ to $RT[1]$.

Routing tables

RTs	$(dest, next, seq, hops)$	
$RT[S]$		$\emptyset$
$RT[1]$	$(S, S, 0, 1)$	$e^1_{(req,S)}$
	$(D, D, 0, 1)$	$e^1_{(rep,D)}$
$RT[2]$	$(S, S, 0, 1)$	$e^1_{(req,S)}$
$RT[D]$	$(S, 1, 0, 2)$	$e^1_{(req,1)}$

(b)-(5): Node 1 sends an $rrep = (S, 0, D, 1, 2)$ to node S which is the *next* node of entry $(S, S, 0, 1)$. Node S adds an entry for D after receiving the *rrep* from node 1. As a result, the route between S and D has established.

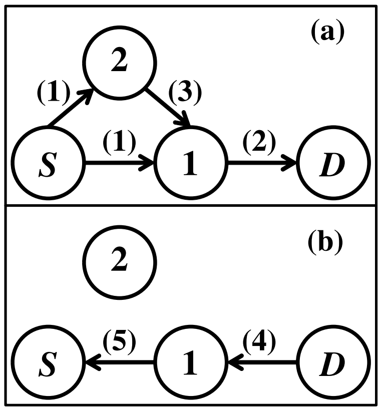

Figure 2: Route establishment process

Routing tables

RTs	$(dest, next, seq, hops)$	
$RT[S]$	$(D, 1, 0, 2)$	$e^1_{(rep,1)}$
$RT[1]$	$(S, S, 0, 1)$	$e^1_{(req,S)}$
	$(D, D, 0, 1)$	$e^1_{(rep,D)}$
$RT[2]$	$(S, S, 0, 1)$	$e^1_{(req,S)}$
$RT[D]$	$(S, 1, 0, 2)$	$e^1_{(req,1)}$

3 SPIN Model

In this section, we show how to represent a MANET protocol using Promela, the input language of the SPIN model checker. Promela has a C-like language syntax. In a Promela program, the system under verification is modeled as a set of concurrent processes. We model a MANET with n nodes using n processes. A process p_{id} has id which is a positive integer value.

3.1 Channel

A process p_{id} has channel CH[id] defined as follows.

```
typedef packet{byte type,sndr,next,src,dest, hops; int seq};
chan CH[NODES] = [BUFF] of {packet};
```

`BUFF` indicates a buffer size of the channel. We set the value of `BUFF` to one. The channel is used to store an incoming message sent by a neighbor node.

3.2 Routing Table

A process p_{id} has a routing table RT[id] that is defined below. RT_LENGTH represents the maximum number of entries in RT[id].

```
typedef entry{byte seq,dest,next,hops};
typedef routing_table{entry entries[RT_LENGTH]};
routing_table table RT[NODES];
```

Table 2: Combinations of selected channels

	selected channels		selected channels
cs1:	{`CH[2]`, `CH[3]`}	cs7:	{`CH[3]`}
cs2:	{`CH[2]`, `CH[4]`}	cs8:	{`CH[4]`, `CH[5]`}
cs3:	{`CH[2]`, `CH[5]`}	cs9:	{`CH[4]`}
cs4:	{`CH[2]`}	cs10:	{`CH[5]`}
cs5:	{`CH[3]`, `CH[4]`}	cs11:	{}
cs6:	{`CH[3]`, `CH[5]`}		

3.3 Broadcasting and Mobility

We have to represent node movements and broadcast of packets. Nodes that receive packets broadcasted by a sender node must be in sender node's communication range. Fig. 3 shows a Promela code that represents broadcasting and node movements simultaneously. Here the number of nodes in the MANET is assumed to be five. This code uses the nondeterministic **do** construct of Promela to nondeterministically select some receiver nodes. The message broadcasted is replicated and put into the incoming channels of these receiver nodes.

This receiver node selection is performed as follows. Array sent[], where all entries are initially 0, is used to represent the receiver nodes already selected. Line 1 is used to prevent from choosing the sender process p_{id} as a receiver node. Line 4 to line 9 are used to nondeterministically select a node (lines 4 to 8) or decide not to select a node (line 9). Line 12 to line 19 are used to send a packet. If the channel of a receiver node, denoted as p_{index}, is full, then the older packet in the channel is removed and then the new broadcasted packet is put into the channel. In this case p_{index} will never receive the discarded old packet.

Parameter $AROUND$ specifies the maximum number of receiver nodes selected when a message is broadcasted. A small $AROUND$ value restricts possible topologies and possible dynamic topology changes but reduces the state space and in turn time and memory used by model checking.

In more detail, by enforcing the process to nondeterministically select channels of up to $AROUND$ receiver nodes, the Promela code in Fig. 3 represents all possible topologies with every node having at most $AROUND$ neighbors. When $AROUND$ is two, the channels are selected at most twice and thus the possible combinations of channels that process p_1 can select become as listed in Table 2. For example, if line 5 and line 6 are nondeterministically selected to execute during the two iterations of the do-loop, then the selected channels will be `CH[2]` and `CH[3]` (cs1 in Table 2), in which case p_1 will put a broadcast packet into both `CH[2]` and `CH[3]`. This models the case where there are two neighbor nodes (node 2 and node 3) in the propagation range of p_1. Combinations cs4, cs7, cs9 and cs10 occur if line 9 is executed in one iteration and one of line 5 to line 8 is executed in the other iteration. These combinations mean that the number of neighbor nodes of p_1 is one. There can be no neighbor node around p_1 and this will happen when line 9 is selected twice (cs11). Note that all these possibilities summarized in Table 2 are examined when model checking is performed and that dynamic changes in topology due to node mobility can be represented as long as the number of neighbor nodes does not exceed $AROUND$.

4 Proposed Technique

We consider the problem of model checking the whole process of route establishment initiated by a single source node. In this section we present our technique to reduce the state space of the model checking problem. In AODV, in face of failures the source node repeats the route establishment process at most three times. As we will show later, the whole series of these trials induces a very large state space and makes the amount of memory and time required prohibitively large. In this section we propose a modeling technique to mitigate state space explosion by reducing the state space.

```
sent[id]=1;
cnt=0;
do::(cnt < AROUND)→
  if::(sent[1]==0)→index=1;
    ::(sent[2]==0)→index=2;
    ::(sent[3]==0)→index=3;
    ::(sent[4]==0)→index=4;
    ::(sent[5]==0)→index=5;
    ::index=255;
  fi;
  cnt++;
  if::(index==255)→skip;
    ::else→sent[index]=1;
    if::(len(CH[index])==BUFF)→
        CH[index]?tmp;
      ::else→skip;
    fi;
    CH[index]!pkt;
  fi;
  ::else→break;
od;
```

Figure 3: Promela code for modeling broadcast and node mobility

Our proposed method focuses on the third trial of route establishment. Let $rreq_S^n$ represents the $rreq$ that the source node broadcasts at the nth round.

Fig. 4 schematically shows how states in the state space are reached as the trials proceed. In Fig. 4, $S^n(n = 1, 2, 3)$ is a set of states at the beginning of the n-th trial. In the following of the section, we show S^3 subsumes S^1 and S^2 and therefore model checking of the third trial, starting with S^3, can detects that can manifest themselves during the third trial as well as those for the first and second trials.

We make the following assumptions.

- No entry in a routing table is discarded because of time out.
- There is exactly one source-destination pair that attempts route establishment.

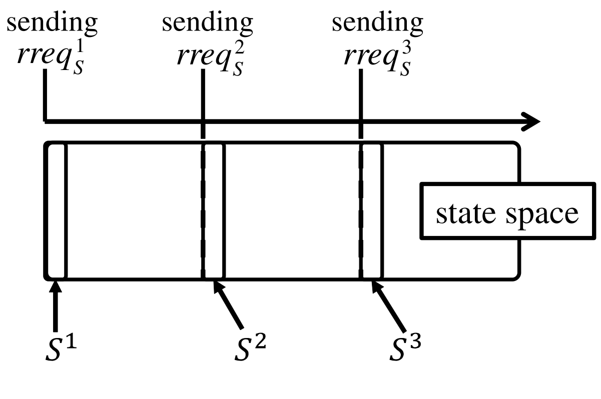

Figure 4: Schematic of state space

Table 3: Routing table entries and received packets

received packet	generated entry
$rreq_i^n$ $(S, i, D, seq_S, seq_D, hops)$	$e^n_{(req,i)}$ $(S, i, seq_S, hops)$
$rrep_j^n$ $(D, j, S, seq_D, hops)$	$e^n_{(rep,j)}$ $(D, j, seq_D, hops)$

- Routing tables initially has no entries relevant to the route establishment.

We show that state that can occur in the first and second trials also occur in the third trial. Our argument is as follows. In any round, a node executes an action only when a packet is received (line 9 of Fig. 1). The number of the current trial, which is reflected in the seq field of the packet, may enable the actions of lines 12, 14, and 16 if the trail number is greater than that of the corresponding table entry (e1.seq < pkt.seq in line 12, for example). On the other hand, if the trail number is smaller than or equal to that of the entry, then it never disables any actions. This means that packets with a lower trail number than the current trail number cannot produce new states of the routing table if the current entries of the routing table are the same. In the following subsections, we show that the set of initial states of the third trial S_3 contains the states in S_1 and S_2; thus checking the third trial suffices to produce possible states.

4.1 Classification of the State of a Routing Table

As we assume that there is a single pair of source and destination nodes, there are at most two entries that are relevant to the model checking problem, more specifically, the entries whose dest (the first element) is the source or destination node. Table 3 shows symbolic representation of these two entries. Note that an entry is generated or updated in response to receipt of a packet. The left column shows the packets that yield the entries on the right column. Here n represents the trial number whose range is $n \in \{1, 2, 3\}$. i and j represent the sender node.

In the rest of the section we show that the entry set of a routing table at states in S^3 can be classified into six patterns as shown below.

$$\begin{array}{lll} rt1{:}\emptyset & rt2{:}\{e^1_{(req,i)}\} & rt3{:}\{e^1_{(req,i)}, e^1_{(rep,j)}\} \\ rt4{:}\{e^2_{(req,i)}\} & rt5{:}\{e^2_{(req,i)}, e^1_{(rep,j)}\} & rt6{:}\{e^2_{(req,i)}, e^2_{(rep,j)}\} \end{array}$$

$rt1 : \emptyset$

The case $rt1$ means that no packet relevant to the source-destination pair of interest has been received until the beginning of the third trial. The entries in the routing table rt satisfy:

$$\forall e \in rt : e.dest \neq rreq^n_S.src \tag{11}$$

$rt2 : \{e^1_{(req,i)}\}$

This case $rt2$ means that the node received $rreq^1_i$ at least once but no $rrep$ until the beginning of the third trial. In this case the routing table rt satisfies:

$$\exists e \in rt : e.seq = rreq^1_i.seq_S \wedge e.dest = rreq^1_i.src \tag{12}$$

$rt3 : \{e^1_{(req,i)}, e^1_{(rep,j)}\}$

This case occurs when the node received $rreq^1_i$ and $rrep^1_j$ at least once. The routing table rt satisfies:

$$\begin{aligned} &\exists e_k, e_l (\neq e_k) \in rt : \\ &e_k.seq = rreq^1_i.seq_S \wedge e_k.dest = rreq^1_i.src \wedge \\ &e_l.seq = rrep^1_j.seq_D \wedge e_l.dest = rrep^1_j.src \end{aligned} \tag{13}$$

$rt4 : \{e^2_{(req,i)}\}$
The case $rt4$ occurs when the node received $rreq^2_i$ at least once and no $rrep$ until the beginning of the third trial. In this case the routing table rt satisfies:

$$\exists e \in rt : e.seq = rreq^2_i.seq_S \land e.dest = rreq^2_i.src \tag{14}$$

$rt5 : \{e^2_{(req,i)}, e^1_{(rep,j)}\}$
This case occurs when the routing table of the node had been in pattern $rt3$ and then the node received $rreq^2_i$ at least once. The routing table satisfies:

$$\begin{aligned}&\exists e_k, e_l(\neq e_k) \in rt :\\ &e_k.seq = rreq^2_i.seq_S \land e_k.dest = rreq^2_i.src \land\\ &e_l.seq = rrep^1_j.seq_D \land e_l.dest = rrep^1_j.src\end{aligned} \tag{15}$$

$rt6 : \{e^2_{(req,i)}, e^2_{(rep,j)}\}$
The case $rt6$ occurs when the routing table of the node had been in pattern $rt4$ and then the node received $rrep^2_j$ at least once. The routing table satisfies:

$$\begin{aligned}&\exists e_k, e_l(\neq e_k) \in rt :\\ &e_k.seq = rreq^2_i.seq_S \land e_k.dest = rreq^2_i.src \land\\ &e_l.seq = rrep^2_j.seq_D \land e_l.dest = rrep^2_j.src\end{aligned} \tag{16}$$

Note that these patterns subsume those that can occur at the beginning of the first or second trials. For example, the initial state of a routing table in the second trial is of either of $rt1$, $rt2$ or $rt3$.

Below we elaborate a more detailed argument about how these patterns occur. First we consider forwarding nodes. Table 4 shows how the routing table of a forwarding node changes during the first round. The column rts shows the entries in the routing table at the beginning of the first round. Initially, there is no entry in the routing table (denoted by $rt1$). The column packet represents the received packet during the first round. When the node receives $rreq^1_i$, the action at line 11 in Fig. 1 is executed. As a result, the routing table becomes $rt2$. After being updated to $rt2$, if the node receives $rrep^1_j$, then the action at line 15 in Fig. 1 is executed, resulting in $rt3$ of the routing table.

Table 5 shows how the routing table is updated in the second round. At the beginning of the second round, the state of the routing stable is either of $rt1$, $rt2$ or $rt3$, as indicated in the rts column of Table 5. Note that $rt1$ occurs when no packet is received in the first round. In the second round, if the node receives $rreq^2_i$, then $rt1$, $rt2$ and $rt3$ become $rt4$, $rt4$ and $rt5$, respectively. When the node receives $rrep^2_j$ after having received $rreq^2_i$, $rt4$ and $rt5$ both transit to $rt6$. If the node receives no packet during the second round, its routing table stays the same, meaning that the state is either $rt1$, $rt2$ or $rt3$.

Table 6 shows the updates in the third round. The column rts in Table 6 shows the possible six states at the beginning of the third round. Note that the set of these states subsumes the set of the initial states of the first round (Table 4) and that of the second round (Table 5).

Similarly, we summarize the changes of the routing table of the destination node in Table 7, Table 8 and Table 9. As can be seen in these tables, the routing table of the destination node is either $rt1$, $rt2$ or $rt4$ at the beginning of the third round.

4.2 Effects of In-Transit Packets

Although the routing table at the beginning of the third trial is of either one of the six patterns as shown above, there can be in-transit packets, such as $rreq^1_i$, $rreq^2_i$, $rrep^1_j$ and $rrep^2_j$, and if a node receives such a packet, the state of the routing table can be changed. Here we show that the presence of these in-transit packets does not affect our argument that the state of a routing table is of either one of these six patterns.

Table 4: Updates of forwarding node's routing table in the first round

rts	packet	action	entry	packet	action	entry
$rt1(\emptyset)$	$rreq_i^1$	Line 11	$rt2(\{e^1_{(req,i)}\})$	$rrep_j^1$	Line 15	$rt3(\{e^1_{(req,i)}, e^1_{(rep,j)}\})$

Table 5: Updates of forwarding node's routing table in the second round

rts	packet	action	entry	packet	action	entry
$rt1(\emptyset)$	$rreq_i^2$	Line 11	$rt4(\{e^2_{(req,i)}\})$	$rrep_j^2$	Line 15	$rt6(\{e^2_{(req,i)}, e^2_{(rep,j)}\})$
$rt2(\{e^1_{(req,i)}\})$	$rreq_i^2$	Line 12	$rt4(\{e^2_{(req,i)}\})$	$rrep_j^2$	Line 15	$rt6(\{e^2_{(req,i)}, e^2_{(rep,j)}\})$
$rt3(\{e^1_{(req,i)}, e^1_{(rep,j)}\})$	$rreq_i^2$	Line 12	$rt5(\{e^2_{(req,i)}, e^1_{(rep,j)}\})$	$rrep_j^2$	Line 16	$rt6(\{e^2_{(req,i)}, e^2_{(rep,j)}\})$

Table 6: Updates of forwarding node's routing table in the third round

rts	packet	action	entry	packet	action	entry
$rt1(\emptyset)$	$rreq_i^3$	Line 11	$\{e^3_{(req,i)}\}$	$rrep_j^3$	Line 15	$\{e^3_{(req,i)}, e^3_{(rep,j)}\}$
$rt2(\{e^1_{(req,i)}\})$	$rreq_i^3$	Line 12	$\{e^3_{(req,i)}\}$	$rrep_j^3$	Line 15	$\{e^3_{(req,i)}, e^3_{(rep,j)}\}$
$rt3(\{e^1_{(req,i)}, e^1_{(rep,j)}\})$	$rreq_i^3$	Line 12	$\{e^3_{(req,i)}, e^1_{(rep,j)}\}$	$rrep_j^3$	Line 16	$\{e^3_{(req,i)}, e^3_{(rep,j)}\}$
$rt4(\{e^2_{(req,i)}\})$	$rreq_i^3$	Line 12	$\{e^3_{(req,i)}\}$	$rrep_j^3$	Line 15	$\{e^3_{(req,i)}, e^3_{(rep,j)}\}$
$rt5(\{e^2_{(req,i)}, e^1_{(rep,j)}\})$	$rreq_i^3$	Line 11	$\{e^3_{(req,i)}, e^1_{(rep,j)}\}$	$rrep_j^3$	Line 16	$\{e^3_{(req,i)}, e^3_{(rep,j)}\}$
$rt6(\{e^2_{(req,i)}, e^2_{(rep,j)}\})$	$rreq_i^3$	Line 11	$\{e^3_{(req,i)}, e^2_{(rep,j)}\}$	$rrep_j^3$	Line 16	$\{e^3_{(req,i)}, e^3_{(rep,j)}\}$

Table 10 summarizes the effects of receipt of an in-transit packet on the routing table. In this table the left column shows the classified type of the routing table before receiving the packet. The middle column shows received packets sent in the first or second trial. The right column shows the pattern of the routing tables after receiving the packet. As can be seen in the table, even if a node receives a packet sent in the previous trials, the routing table is still of one of the six patterns. Therefore we can ignore in-transit packets and can assume that the channels are empty when the third trial starts.

4.3 Generating SPIN Models with Different Initial States

To model check all different initial states of the third trial using the SPIN model checker, we create instances of a SPIN model each with a different initial state. As there is a total of six possible patterns for a routing table, there are 6^m initial states where m is the number of nodes in the MANET.

Table 7: Updates of destination node's routing table in the first round

rts	packet	action	entry
$rt1(\emptyset)$	$rreq_i^1$	Line 13	$\{e^1_{(req,i)}\}$

Table 8: Updates of destination node's routing table in the second round

rts	packet	action	entry
$rt1(\emptyset)$	$rreq_i^2$	Line 13	$\{e^2_{(req,i)}\}$
$rt2(\{e^1_{(req,i)}\})$	$rreq_i^2$	Line 14	$\{e^2_{(req,i)}\}$

Table 9: Updates of destination node's routing table in the third round

rts	packet	action	entry
$rt1(\emptyset)$	$rreq_i^3$	Line 13	$\{e^3_{(req,i)}\}$
$rt2(\{e^1_{(req,i)}\})$	$rreq_i^3$	Line 14	$\{e^3_{(req,i)}\}$
$rt4(\{e^2_{(req,i)}\})$	$rreq_i^3$	Line 14	$\{e^3_{(req,i)}\}$

Table 10: Routing table patterns before and after receiving a packet

entries before receiving packets	received packet	entries after receiving packets
$rt1 : \emptyset$	$rreq_i^1$	$rt2 : \{e^1_{(req,i)}\}$
	$rrep_j^1$	$rt1 : \emptyset$
	$rreq_i^2$	$rt4 : \{e^2_{(req,i)}\}$
	$rrep_j^2$	$rt1 : \emptyset$
$rt2 : \{e^1_{(req,i)}\}$	$rreq_i^1$	$rt2 : \{e^1_{(req,i)}\}$
	$rrep_j^1$	$rt3 : \{e^1_{(req,i)}, e^1_{(rep,j)}\}$
	$rreq_i^2$	$rt4 : \{e^2_{(req,i)}\}$
	$rrep_j^2$	$rt3 : \{e^1_{(req,i)}, e^1_{(rep,j)}\}$
$rt3 : \{e^1_{(req,i)}, e^1_{(rep,j)}\}$	$rreq_i^1$	$rt3 : \{e^1_{(req,i)}, e^1_{(rep,j)}\}$
	$rrep_j^1$	$rt3 : \{e^1_{(req,i)}, e^1_{(rep,j)}\}$
	$rreq_i^2$	$rt4 : \{e^2_{(req,i)}, e^1_{(rep,j)}\}$
	$rrep_j^2$	$rt3 : \{e^1_{(req,i)}, e^1_{(rep,j)}\}$
$rt4 : \{e^2_{(req,i)}\}$	$rreq_i^1$	$rt4 : \{e^2_{(req,i)}\}$
	$rrep_j^1$	$rt5 : \{e^2_{(req,i)}, e^1_{(rep,j)}\}$
	$rreq_i^2$	$rt4 : \{e^2_{(req,i)}\}$
	$rrep_j^2$	$rt6 : \{e^2_{(req,i)}, e^2_{(rep,j)}\}$
$rt5 : \{e^2_{(req,i)}, e^1_{(rep,j)}\}$	$rreq_i^1$	$rt5 : \{e^2_{(req,i)}, e^1_{(rep,j)}\}$
	$rrep_j^1$	$rt5 : \{e^2_{(req,i)}, e^1_{(rep,j)}\}$
	$rreq_i^2$	$rt5 : \{e^2_{(req,i)}, e^1_{(rep,j)}\}$
	$rrep_j^2$	$rt6 : \{e^2_{(req,i)}, e^2_{(rep,j)}\}$
$rt6 : \{e^2_{(req,i)}, e^2_{(rep,j)}\}$	$rreq_i^1$	$rt6 : \{e^2_{(req,i)}, e^2_{(rep,j)}\}$
	$rrep_j^1$	$rt6 : \{e^2_{(req,i)}, e^2_{(rep,j)}\}$
	$rreq_i^2$	$rt6 : \{e^2_{(req,i)}, e^2_{(rep,j)}\}$
	$rrep_j^2$	$rt6 : \{e^2_{(req,i)}, e^2_{(rep,j)}\}$

This number can be reduced based on the following two observations. First, the routing table of the source node can be ignored since once having sent the *rreq*, the source node is not involved in route establishment until it receives the *rrep*. Second, although the details are omitted, it can be shown that the routing table of the destination node has only three possible patterns. As a result, the total number of SPIN model instances is $6^{m-2} \times 3$. When $m = 4$, this number becomes 108.

5 Experimental Result

In this section we show the results of two experiments. The first experiment evaluates the effectiveness of the proposed modeling technique with respect to performance of model checking. The second experiment is intended to demonstrate the ability of detecting design faults.

5.1 Performance Evaluation

The purpose of this experiment is to evaluate the effectiveness of the proposed technique with respect to model checking performance. In this experiment we considered MANETs consisting of four nodes and five nodes and assumed that each node has at most two neighbor nodes at any time instance. Of the four or five nodes, two particular nodes were designated as the source node and the destination node. The machine specifications and the model checking tool are as follows.

Table 11: Experimental results (two nodes)

	proposed technique	full state space search
Execution time[s]	3.0×10^{-2}	$< 1.0 \times 10^{-2}$
State space	99	79
Memory usage[MB]	3.8×10^{3}	1.3×10^{3}

Table 12: Experimental results (three nodes)

	proposed technique	full state space search
Execution time[s]	2.0×10^{-1}	3.0×10^{-1}
State space	3.4×10^{4}	3.4×10^{4}
Memory usage[MB]	2.3×10^{3}	1.4×10^{3}

Table 13: Experimental results (four nodes)

	proposed technique	full state space search
Execution time[s]	6.7×10	$> 2.2 \times 10^{3}$
State space	2.7×10^{7}	$> 4.1 \times 10^{8}$
Memory usage[MB]	2.1×10^{4}	$> 1.0 \times 10^{6}$

Table 14: Experimental results (five nodes)

	proposed technique	full state space search
Execution time[s]	5.6×10^{4}	$> 2.3 \times 10^{3}$
State space	6.7×10^{10}	$> 3.1 \times 10^{8}$
Memory usage[MB]	2.3×10^{7}	$> 1.0 \times 10^{6}$

OS : CentOS 6.6

CPU : Xeon E5-2665

MEM : 128GB

Tool : SPIN 6.2.5

For the four node network, as described in the previous section, we created 108 ($6^2 \times 3$) SPIN models each with a different initial state (combination of patterns) of routing tables. For the five node network, we created 648 ($6^3 \times 3$) SPIN models. The execution time is the summation of the execution time required for model checking all the instances. The size of the state space and the amount of memory used are also the summation for all the instances.

For comparison purposes, we also measured performance of full state space search, that is, model checking of the whole series of the three trials from the beginning. The results of the experiment are summarized in Tables 11, 12, 13 and 14. We varied the number of nodes from two to four and each table shows the results for each of these four cases.

When the number of nodes was two or three, model checking was finished almost instantly for both approaches. Interestingly the full state space search yielded a smaller state space for the two node-case. This can be explained as follows. The full state space search does not explore the same state more than once. On the other hand, the proposed approach executes multiple instances of model checking and the same state may be visited in two or more runs.

When the number of nodes was three or four, the full state space search was not completed because available memory was exhausted. On the other hand, with our proposed technique, model checking was successfully completed for both all the 108 instances and all the 648 instances. Note that the number of states explored by the full state space search until memory exhaustion was much larger than that of states explored by all runs of the proposed approach. This and the results for fewer nodes show that reachable states very rapidly increase as the number of nodes grows and that the proposed approach can significantly lower the increase by achieving considerable state space reduction. Execution time, state space, and memory usage were all reduced significantly as shown in Tables 13 and 14.

No defects were found in the experiments. This is unsurprising as AODV is time tested.

Each of the SPIN models generated by the proposed technique can be executed in isolation, because these models are independent with each other. This is why the proposed techniques were successful in completing verification although the total time and memory usage was larger than the ordinary model checking case.

5.2 Loop Detection

It is known that an older version of AODV had a defect which causes loop generation. For example, in [6], some scenarios where loop routes are generated were detected by applying model checking and theorem proving to a MANET with three-node linear topologies. Fig. 5 illustrates one such scenario where:

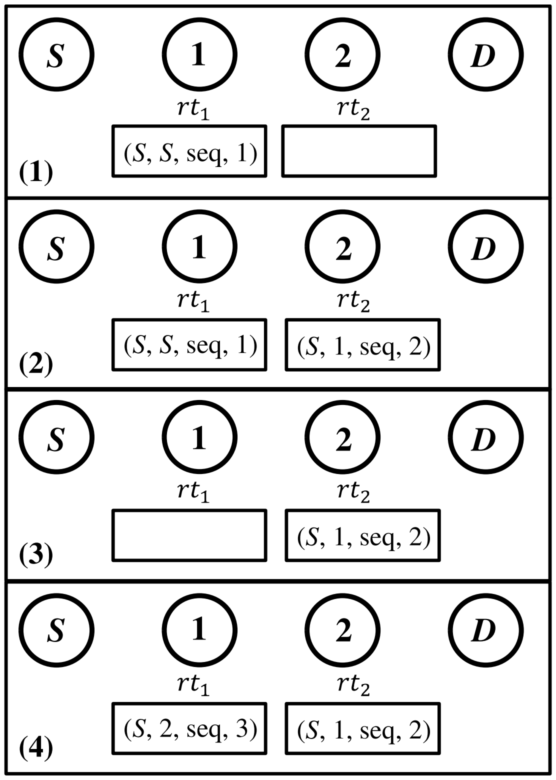

Figure 5: Example of a loop occurrence

Fig. 5(1) : Node 1 receives *rreq* from S.
Fig. 5(2) : Node 1 broadcasts *rreq*, then node 2 receives it.
Fig. 5(3) : The routing table entry of node 1 is discarded.
Fig. 5(4) : Node 2 broadcasts *rreq*, then node 1 receives it.
Afterwards, the entry of node 1 enforces it to send packets whose destination is S to node 2. Node 2 sends a packet whose destination is S to node 1, thus resulting in a loop.

Aiming to show that our technique can be useful in detecting a defect, we conducted an experiment where we attempted to detect loop. As it was known that this error occurs only when routing table entries are discarded, we extended our model such that a node may discard the entries because of time out. In order to detect a loop, we embedded C language code into the SPIN model written in Promela. By consulting the routing tables of the nodes, this code traverses a route from

Table 15: Experimental results for loop detection

	four nodes	five nodes
Execution time[s]	5.6×10	5.8×10^2
State space	3.7×10^7	9.0×10^8
Memory usage[MB]	2.3×10^4	3.7×10^5

```
 1:  c_code{
 2:   int cnt=0,id=PNode→id;
 3:   now.loop=0;
 4:   while(cnt<RT_LENGTH){
 5:    if(now.RT[id].entries[cnt].dest==S){
 6:     id=now.RT[id].entries[cnt].next;
 7:     if(id==S){break;}
 8:     else if(id==PNode→id){
 9:      now.loop=1;break;}
10:     else{cnt=0;}
11:    }
12:    else{cnt++;}
13:   }
14:  };
15:  assert(loop==0);
```

Figure 6: Embedded C code for loop detection

a given node to the source node. If the node appears on the way to the source node, then it can be concluded that the traced route forms a loop.

Fig. 6 shows the embedded C code (lines 1–14) and an assertion (line 15) for detecting the existence of a loop. This code is executed by every process (i.e., node). PNode→ id is the ID of the node that executes the C code; thus id is initially set to the ID of the node (line 2). The keyword now is used to access Promela variables from C code. The variable loop is a flag in the Promela code to decide the existence of a loop and is initialized and set in the C code by setting now.loop to 0 (line 3) and 1 (line 9).

Route traverse is carried out as follows. First the routing table of node *id* is checked to see if there is an entry for S. The while loop (lines 4–13) is used for this purpose, where cnt, ranging from 0 to RT_LENGTH -1, is used as the index for entries. The keyword now is used to access the routing table: the cnt-th entry can be read by accessing now.RT[id].entries[cnt]. If there is an entry for destination S (line=5), then id is updated to the next node on the route (line 6). If the next node is S, then the traverse on the route to S finishes without encountering a loop, thus finishing loop detection by breaking the while loop (line 7). If the next node is PNode→ id, i.e., the node that initiates loop detection, then it turns out that a loop exists (line 9). If the above two cases do not apply, then route traversal continues by checking the route table of node id from the first entry.

Table 15 show the results of experiments on loop detection. The execution time is the summation of the execution time required for model checking all the instances. The size of the state space and the amount of memory used are also the summation for all the instances. We created a total of 108 instances and of 648 instances of the new SPIN model with the loop detection code. For all these instances, the possibility of a loop occurrence was successfully detected by SPIN.

6 Related Work

SPIN model checker

SPIN is the most commonly used model checker. Model checking of AODV with SPIN is described in [6], where model checking was used to show that a loop can be established in an older version of AODV. In [14], De Renesse et al. describe their attempt to model check the WARP protocol. In their attempt, it is assumed that a network consists of five nodes and that the neighbor nodes of two of the five nodes are not changed. Model checking was used to check reachability of packets between two nodes by using *supertrace/bitstate* mode at runtime option with SPIN. In [36], Wibling et al. apply SPIN and the UPPAAL model checker to the LUNAR protocol. When using SPIN they considered six different topologies and verified that a message from a source node reaches the destination node. Using UPPAAL, they considered eight topologies and verified the protocol against deadlock freedom, route establishment and message reachability. Câmara et al. proposed techniques to model protocol's behavior as two models [8, 9]: One model represents the internal behavior of a node which concerns processing of received packets, whereas the other model represents the external behavior model that concerns packet transfer between nodes. They showed that their modeling techniques were capable of detecting known defects of some MANET protocols, including LAR [27], DREAM [5] and OLSR. In [35], Steele et al. used to SPIN to verify OLSR against the property that a route exists for every destination node. Using SPIN they also showed some topologies where a node suffering from Byzantine failures cannot be detected.

UPPAAL model checker

UPPAAL is a popular model checker which has often been used for MANET verification. A distinguishing feature of UPPAAL is that it can verify real-time properties. In [10], Chiyangwa et al. model checked AODV with UPPAAL. They showed that AODV with recommended timing parameter settings may result in a failure of route establishment and proposed how to solve this problem. In [37], LUNAR was model checked against properties such as route establishment and message reachability. In [17, 15], Fehnker et al. used model checking to demonstrate that the optimal route is not always established in AODV. In [23] and [12], SMC–UPPAAL [13], a statistical extension of the UPPAAL, was used for model checking of AODV and the DYMO protocol [34].

Others

Process algebras have been used for modeling and verification of MANET protocols. In [38], Zakiuddin et al. used CSP [22] to model CBRP (Cluster Based Routing Protocol) and verified the correctness with FDR [21], a CSP model checker. Some process algebras for MANETs have been proposed in [38, 18, 30, 19, 20]. These algebras did not come with a model checker. Exceptionally in [24], Höfner et al. used a process algebra called AWN [16] to model a protocol and model checked it by translating the model into UPPAAL.

A distinguishing work on MANET protocol verification is the one by Musuvathi et al. [31], where CMC [31], a model checker which works directly on C language source codes, was applied to three implementations of AODV, namely, AODV-UU [1], kernel-AODV [2] and mad-hoc [3]. CMC detected several defects in these source codes.

7 Conclusion

In this paper we proposed model checking techniques that can be used to verify a reactive MANET protocol, namely, AODV. In AODV or other reactive protocols, a source node retries the route establishing process some fixed number of times in face of failures. The proposed techniques allow verification of the whole series of trials only by model checking the last trial, thus resulting in state space reduction. We showed how the SPIN model checker can be used with our proposed technique. The results of experiments showed that the proposed techniques can significantly reduce time and space used in the model checking process and that this reduction may lead to an increase of the number of nodes that can be dealt with.

Also we demonstrated the model developed based on the proposed techniques is capable of finding a protocol design defect. In future, we plan to apply the proposed technique to other MANET protocols, including not only other reactive protocols but also multipath routing protocols. Although this paper only considered AODV, we believe that our approach can be extended to verify these protocols as they often share features in common with AODV. Such protocols include, for example, Multicast Ad hoc On-Demand Distance Vector (MAODV).

Finally we make some remarks on other possible extension of the proposed approach. Although the focus of this paper was limited to route discovery, it should be worth verifying the behavior in other processes. For example, error handling, where *rerr* packets which were ignored in this paper are involved, is as important as route discovery; thus verification against its correctness deserves further study. Our interest in the paper was placed on reactive MANET protocols which rely on on-demand route discovery. As mentioned in the introduction, there is another type of protocols called proactive protocols which do not use on-demand route discovery but periodically update routing tables to maintain the network topology. Verification of this type of protocols should also be addressed. One possible direction of extending the proposed approach to address this problem is to apply it to verification of a starting phase where a node fulfills empty entries of the routing table by discovering routes to other nodes.

References

[1] *AODV-UU.* http://sourceforge.net/projects/aodvuu/.

[2] *kernel-AODV.* http://w3.antd.nist.gov/wctg/aodv_kernel/.

[3] *mad-hoc.* http://web.archive.org/web/20010401143715/http://mad-hoc.flyinglinux.net/.

[4] E. Alotaibi and B. Mukherjee. A survey on routing algorithms for wireless ad-hoc and mesh networks. *Computer Networks*, 56(2):940 – 965, 2012.

[5] S. Basagni, I. Chlamtac, V.R. Syrotiuk, and B.A. Woodward. A distance routing effect algorithm for mobility (dream). In *Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, MobiCom '98, pages 76–84, New York, NY, USA, 1998. ACM.

[6] K. Bhargavan, D. Obradovic, and C.A. Gunter. Formal verification of standards for distance vector routing protocols. *J. ACM*, 49(4):538–576, July 2002.

[7] A. Boukerche, B. Turgut, N. Aydin, M.Z. Ahmad, L. Bölöni, and D. Turgut. Routing protocols in ad hoc networks: A survey. *Computer Networks*, 55(13):3032 – 3080, 2011.

[8] D. Câmara, A.A.F. Loureiro, and F. Filali. Methodology for formal verification of routing protocols for ad hoc wireless networks. In *Proceedings of the Global Telecommunications Conference, IEEE GLOBECOM '07*, pages 705–709, Nov 2007.

[9] D. Câmara, A.A.F. Loureiro, and F. Filali. Formal verification of routing protocols for wireless ad hoc networks. In *Guide to Wireless Ad Hoc Networks*, Computer Communications and Networks, pages 189–210. Springer London, 2009.

[10] S. Chiyangwa and M. Kwiatkowska. A timing analysis of aodv. *Formal Methods for Open Object-Based Distributed Systems, Lecture Notes in Computer Science*, 3535:306–321, 2005.

[11] T. Clausen and P. Jacquet. *Optimized Link State Routing Protocol (OLSR).* IETF RFC3626, October 2003.

[12] A.D. Corso, D. Macedonio, and M. Merro. Statistical model checking of ad hoc routing protocols in lossy grid networks. *NASA Formal Methods, Lecture Notes in Computer Science*, 9058:112–126, 2015.

[13] A. David, K.G. Larsen, A. Legay, M. Mikučionis, and Z. Wang. Time for statistical model checking of real-time systems. *Computer Aided Verification, Lecture Notes in Computer Science*, 6806:349–355, 2011.

[14] R. de Renesse and A.H. Aghvami. Formal verification of ad-hoc routing protocols using spin model checker. In *Proceedings of the 12th IEEE Mediterranean Electrotechnical Conference 2004*, volume 3, pages 1177–1182, May 2004.

[15] A. Fehnker, R. van Glabbeek, P. Höfner, A. McIver, M. Portmann, and W.L. Tan. Automated analysis of aodv using uppaal. *Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science*, 7214:173–187, 2012.

[16] A. Fehnker, R.J. van Glabbeek, P. Höfner, A. McIver, M. Portmann, and W.L. Tan. A process algebra for wireless mesh networks. *Programming Languages and Systems, Lecture Notes in Computer Science*, 7211:295–315, 2012.

[17] A. Fehnker, R.J. van Glabbeek, P. Höfner, A.K. McIver, M. Portmann, and W.L. Tan. Modelling and analysis of aodv in uppaal. In *International Workshop on Rigorous Protocol Engineering*, 2011.

[18] F. Ghassemi, W. Fokkink, and A. Movaghar. Restricted broadcast process theory. In *Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 345–354, Nov 2008.

[19] F. Ghassemi, W. Fokkink, and A. Movaghar. Equational reasoning on ad hoc networks. *Fundamentals of Software Engineering, Lecture Notes in Computer Science*, 5961:113–128, 2010.

[20] F. Ghassemi, M. Talebi, A. Movaghar, and W. Fokkink. Stochastic restricted broadcast process theory. *Computer Performance Engineering, Lecture Notes in Computer Science*, 6977:72–86, 2011.

[21] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A.W. Roscoe. Fdr3 – modern refinement checker for csp. *Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science*, 8413:187–201, 2014.

[22] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[23] P. Höfner and A. McIver. Statistical model checking of wireless mesh routing protocols. *NASA Formal Methods, Lecture Notes in Computer Science*, 7871:322–336, 2013.

[24] P. Höfner, R.J. van Glabbeek, W.L. Tan, M. Portmann, A. McIver, and A. Fehnker. A rigorous analysis of aodv and its variants. In *Proceedings of the 15th ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems, MSWiM '12*, pages 203–212, New York, NY, USA, 2012. ACM.

[25] G.J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[26] D. Johnson, Y. Hu, and D. Maltz. *The Dynamic Source Routing Protocol (DSR) for Mobile Ad Hoc Networks for IPv4.* IETF RFC4728, Feb. 2007.

[27] Y-B. Ko and N.H. Vaidya. Location-aided routing (lar) in mobile ad hoc networks. *Wireless Networks*, 6(4):307–321, 2000.

[28] H. Kojima, Y. Nagashima, and T. Tsuchiya. Model checking techniques for state space reduction in manet protocol verification. In *31st IEEE International Parallel and Distributed Processing Symposium workshop*, pages 509–516, May 2016.

[29] K.G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[30] M. Merro. An observational theory for mobile ad hoc networks (full version). *Information and Computation*, 207(2):194 – 208, 2009. Special issue on Structural Operational Semantics (SOS).

[31] N. Musuvathi, D.Y.W. Park, A. Chou, D.R. Engler, and D.L. Dill. Cmc: A pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev.*, 36(SI):75–88, December 2002.

[32] R. Ogier, M. Lewis, and F. Templin. *Topology Dissemination Based on Reverse Path Forwarding (TBRPF).* IETF RFC3684, February 2004.

[33] C. Perkins, E. Belding-Royer, and S. Das. *Ad hoc On-demand Distance Vector (AODV) Routing.* IETF RFC3561, July 2003.

[34] C. Perkins, S. Ratliff, and J. Dowdell. *Dynamic MANET On-demand (AODVv2) Routing draft-ietf-manet-dymo-26.* IETF, Feb. 2013.

[35] M.F. Steele and T.R. Andel. Modeling the optimized link-state routing protocol for verification. In *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium*, TMS/DEVS '12, pages 35:1–35:8, San Diego, CA, USA, 2012. Society for Computer Simulation International.

[36] O. Wibling, J. Parrow, and A. Pears. Automatized verification of ad hoc routing protocols. *Formal Techniques for Networked and Distributed Systems FORTE 2004, Lecture Notes in Computer Science*, 3235:343–358, 2004.

[37] O. Wibling, J. Parrow, and A. Pears. Ad hoc routing protocol verification through broadcast abstraction. *Formal Techniques for Networked and Distributed Systems FORTE 2005, Lecture Notes in Computer Science*, 3731:128–142, 2005.

[38] I. Zakiuddin, M. Goldsmith, P. Whittaker, and P. Gardiner. A methodology for model-checking ad-hoc networks. *Model Checking Software, Lecture Notes in Computer Science*, 2648:181–196, 2003.