Mojgan Kamali | Mayam Kamali | Luigia Petre

Formally Analyzing Proactive, Distributed Routing

Turku Centre *for* Computer Science

TUCS Technical Report
No 1125, November 2014

Formally Analyzing Proactive, Distributed Routing

Mojgan Kamali
Åbo Akademi University, Department of Computer Science
Joukahaisenkatu 3-5A, 20520 Turku, Finland
mokamali@abo.fi

Mayam Kamali
University of Liverpool, Department of Computer Science
Liverpool L69 3BX, United Kingdom
maryam.kamali@liverpool.ac.uk

Luigia Petre
Åbo Akademi University, Department of Computer Science
Joukahaisenkatu 3-5A, 20520 Turku, Finland
lpetre@abo.fi

TUCS Technical Report
No 1125, November 2014

Abstract

As (network) software is such an omnipresent component of contemporary mission-critical systems, formal analysis is required to provide the necessary certification or at least assurance for these systems' properties. In this paper we focus on modelling and analysing a distributed, proactive routing protocol named Optimised Link State Routing (OLSR), recognised as the standard ad-hoc routing protocol for Wireless Mesh Networks (WMN). WMNs are instrumental in critical systems, such as emergency response networks and smart electrical grids. We employ Uppaal model checking for analysing safety properties of OLSR as well as to point out a case of OLSR malfunctioning.

Keywords: Network Protocol, Protocol Verification, Uppaal Model Checker, Formal Analysis, Distributed Protocol

TUCS Laboratory
Distributed Systems Laboratory

1 Introduction

Routing is at the center of network communication, which in turn, is part of the backbone for numerous mission-critical systems. For instance, smart electrical grids transport (through wires) electrical power from power generators to consumers, but one of their critical 'smart' features consists in controlling, often wirelessly, the dis-connectors in the power grid. We can think of two inter-related networks in this case, the electricity network transporting electrical power and the telecommunication network, controlling the momentary topology of power distribution. Obviously, the telecommunication network is also a consumer of electrical power. Another example of wireless network-based critical systems is that of emergency response networks, when ad-hoc network structures are formed to help resolve various emergencies, such as earthquake aftermaths. In these and other examples, the wireless communication is truly distributed, without depending on any central entity for coordination. In this paper we focus on distributed routing mechanisms in such wireless networks; due to their usage for critical systems, we aim to model and analyse them.

A routing protocol enables node communication in a network by disseminating information enabling the selection of routes. In this way, nodes are able to sent data packets to arbitrary destinations in the network. Clearly, shortcomings in the routing protocol immediately decrease the performance and reliability of the entire network. Wireless Mesh Networks (WMNs) have gained popularity and are increasingly applied in a wide range of application areas. They are self-organising wireless multi-hop networks which provide support for communication without relying on a wired infrastructure [7]. As a consequence, they bear the benefit of rapid and low-cost network deployment. The Optimised Link State Routing (OLSR) protocol [4] is one of the proactive routing protocols, identified as the standard ad-hoc routing protocol by the IETF MANET working group[1]. By distributing control messages throughout the network, proactive protocols maintain a list of all destinations together with routes to them.

Traditionally, common methods used to evaluate and validate network protocols are test-bed experiments and simulation in 'living lab' conditions. However, such analysis is always limited to very few topologies [8]; moreover, when a shortcoming is found, it is often unclear whether the limitation is a consequence of the routing protocol chosen, or of the underlying link layer (the reason is that often both layers are implemented at the same time and that no clear separation is established). In this paper, we abstract away from the underlying link layer; hence a shortcoming found is definitely a problem of the routing protocol.

Another problem w.r.t. routing protocols is that they are usually specified in English prose. Although this makes them easy to understand, it is well-known that textual description contains ambiguities, contradictions and often lacks specific details. As a consequence, this might yield to different interpretations of one

[1]http://datatracker.ietf.org/wg/manet/charter/

specification and to different implementations [10]. In the worst case, implementations of the same routing protocol are even incompatible.

One approach to address these problems is using formal methods in general and model checking in particular. Formal methods provide valuable tools for design, evaluation, and verification of WMN routing protocols; they complement alternatives such as test-bed experiments and simulation. These methods have a great potential on improving the correctness and precision of design and development, as they produce reliable results. Formal methods allow the formal specification of routing protocols and the verification of their desired behaviour by applying mathematics and logics [3]. In this way, stronger and more general assurance about protocol behaviour and properties can be achieved.

The concrete result reported in this paper consists in applying model checking to explore the behaviour of WMN routing protocols. Model checking [3] is a powerful approach used for validating key correctness properties in finite representations of a formal system model [8]. We put forward the applicability of model checking for providing a clear and unambiguous Uppaal [15] model of the OLSR protocol; based on it, we carry out some experiments in order to analyse OLSR behaviour; remarkably, we uncover some problematic behaviour of this protocol. However, we believe these findings are more generally applicable. Namely, distributed control is a topic of high relevance, both theoretically and in practice, the latter due to its potential applicability in self-recovering, distributed systems such as the smart grids. Due to space limitations, we only discuss this briefly in conclusions, pointing out several lines of future research.

We proceed as follows. In Section 2, we overview the OLSR protocol. We detail the Uppaal model of OLSR, based on RFC 3626 [4], in Section 3. In Section 4, we present the results of our experiments. We review related work in Section 5, and propose future directions as well as conclude in Section 6.

2 OLSR Overview

OLSR [4] is a proactive routing protocol particularly designed for WMNs and Mobile Ad hoc Networks (MANETs). The proactive nature of OLSR implies the benefit of having the routes available when needed. The underlying mechanism of this protocol consists in the periodic exchange of messages to find routes. OLSR works in a completely distributed manner without depending on any central entity. As a consequence, it is applicable in situations where a large subset of nodes are communicating with each other or in situations where nodes are changing with time. The protocol minimises flooding of control messages in the network by selecting the so-called Multipoint Relays (MPRs). MPRs are one-hop neighbours of every node which have bi-directional links towards two-hop neighbours of that node [13]. The process of MPR selection is described shortly.

Nodes running OLSR are not restricted to any kind of start up synchronisation.

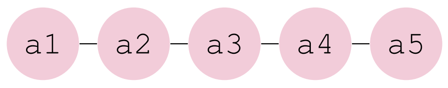

Fig. 1: A network of size 5.

Nodes	a1	a2	a3	a4	a5
MPRs	a2	a3	a2, a4	a3	a4

Table 1: Nodes and their MPRs.

Every node broadcasts a HELLO message every 2 seconds in the network and detects its direct neighbour nodes by receiving these messages. A HELLO message contains information about one-hop neighbours of the originator which allows the receiving nodes to know about their two-hop neighbours. HELLO messages traverse only one wireless link or a single hop and they are not forwarded. This type of message is used for neighbour detection and MPR selection.

After receiving HELLO messages from direct neighbours, every node selects its MPRs and selected MPRs understand about their MPR selectors (those nodes that have selected them as an MPR). Then, MPR nodes broadcast Topology Control (TC) messages every 5 seconds to build and update topological information which can be transmitted on more than one wireless link by intermediate MPR nodes. This means that if a node is not an MPR, it receives TC messages, processes those messages, but it will not forward them. Every TC message contains the information about MPR selectors of the TC originator. While receiving control messages from other nodes, every node updates its routing table for the originator of the received message. After broadcasting and forwarding control messages via nodes, routes to all destinations should be established and nodes should have the required information about all the other nodes in the network. As a consequence, nodes can select paths to deliver data packets to arbitrary destination nodes.

Fig. 1 depicts a network topology consisting of 5 nodes. As shown in Table 1, each node selects its MPRs from its direct neighbours. For instance, node a2 selects node a3 as its MPR because a3 has a link toward two-hop neighbour of a2. In this network, only nodes a2, a3 and a4, as selected MPRs, broadcast TC messages and provide required information for other nodes in the network. Selecting these nodes as TC generators decreases traffic in the network, since not all the nodes broadcast TC.

We describe below the behaviour of nodes a1, a2 and a3 running OLSR to sketch an overview of this protocol. Lets assume node a3 starts first, then node a2 start working, and at last node a1 broadcasts its HELLO message. The first broadcasted HELLO message via node a3 has the following information:

- HELLO is the type of the message.
- 3 is the originator of the message.

This message has no information about the one-hop neighbours or MPRs of this node, since it is the starting point of node `a3` and no information is available in its routing table about other nodes. The HELLO message from node `a3` is received by the immediate neighbours, i.e., nodes `a2` and `a4`. While receiving a HELLO message from node `a3`, node `a2` updates its routing table for this node and then node `a2` is broadcasting its HELLO message which contains the following information:

- HELLO is the type of the message.
- 2 is the originator of the message.
- 3 is one-hop neighbour of node 2.

While receiving a HELLO message from node `a2`, node `a1` updates its routing table for `a2` and `a2`'s one-hop neighbour, i.e., node `a3`. By this, node `a1` learns about its two-hop neighbours and selects node `a2` as its MPR. Then, node `a1` broadcasts a HELLO message and declares node `a2` as its MPR. Upon receiving HELLO from node `a1`, `a2` would figure out that it has been selected as an MPR by `a1`. So, it broadcasts its TC message which has the following information:

- TC is the type of the message.
- 2 is the originator of the message.
- 1 is sequence number of the message.
- 5 is the time to live of the message.
- 0 is the number of hops from the originator of this TC message.
- 2 is the address of the sender.
- 1 is the MPR selector of node `a2`.

This process continues for other nodes in the network.

3 Modelling OLSR in Uppaal

Our larger context goal consists in providing formal mechanisms for the specification, analysis, and comparison of various WMN protocols. In this paper, we use Uppaal [15] to model and investigate the behaviour of the OLSR protocol. Uppaal [1] is a well established model checker for modelling, simulating and verifying real-time systems. It is designed for systems that can be modelled as networks of timed automata, used in particular for protocol verification. We use Uppaal for the following reasons: *a*) two synchronisation mechanisms of wireless networks, i.e., broadcast and binary synchronisation, are provided by Uppaal;

b) Uppaal provides common data structures, such as structs and arrays; a C-like programming language is applicable for defining updates on these data structures; *c*) OLSR highly depends on on-time broadcasting of control messages and Uppaal provides mechanisms and tools for considering time variables. In the following, we describe Uppaal to the extent needed in this paper.

3.1 Uppaal timed-automata

The Uppaal modelling language extends timed automata with various features [1]. Uppaal automata provide various types and data structures, and variables of these types. The system state is defined as the value of all variables, local in some automata or shared between automata. Every automaton is a graph with locations and edges between these locations together with guards and clock constraints. Each location might have an invariant which is a slide-effect free expression; only clock, integer variables, and constants are referenced, and each edge has a selection, a guard, a synchronisation label, and optionally an update. Selection non-deterministically bind a given identifier to a value in a given range. Guards on transitions are used to restrict the automaton behaviour. Synchronisation happens via channels; for every channel a there is one label $a!$ to identify a sender, and $a?$ to represent a receiver. Transitions with no labels are internal transitions and all the other transitions use one of the two following types of synchronisation [1].

In *binary handshake* synchronisation, one automaton which has an edge with a !-label synchronises with another automaton with the edge having a ?-label. These two transitions synchronise only when both guards evaluate to true in the current state. After taking the transitions, both locations will change, and the updates on transitions will be applied to the state variables; first the updates will be done on the !-edge, then the updates occur on the ?-edge. When having more than one possible pair, the transition will be selected non-deterministically [1].

In *broadcast* synchronisation, one automaton with an !-edge synchronies with several other automata that all have an edge with a relevant ?-label. The initiating automaton is able to change its location, and apply its update, if and only if the guard on its edge is satisfied. It does not need a second automaton to synchronise with. Matching ?-edge automata must synchronise if their guard is true, currently. They will change their location and do the updating of the state. At first, the automaton with the !-edge will update the state, then the other automata will follow in some lexicographic order. When more than one automaton can initiate a transition on an !-edge, the process of choosing will occur non-deterministically [1].

Urgent channels are special type of channels which must be taken with no delay. In other words, delays must not happen if a synchronisation transition on an urgent channel is enabled.

Committed locations are special type of locations where delay cannot happen and the next transition outgoing from a committed location must be taken immediately. These locations freeze time; i.e. time is not allowed to pass when a process is in one of them. When a model has one or more active committed locations, no transitions other than those leaving said locations can be enabled. if several processes are in a committed location at the same time, then they will interleave.

Uppaal's verifier uses Computation Tree Logic (CTL) [6] to model system properties. The query language in Uppaal contains two types of formulas, namely path formulas and state formulas. State formulas describe individual states of the model, while path formulas quantify over paths or traces in the model. CTL uses $\mathbf{A}$ and $\mathbf{E}$ as path quantifiers, and $\mathbf{G}, \mathbf{F}, \mathbf{X}$, and $\mathbf{U}$ as temporal operators. Here, a path contains an infinite sequence of states which are connected using transitions.

Formulas model what can happen starting from the 'current' state, meaning the state being described in the formula. The current state is included in its future. In this context $\mathbf{A}\phi$ means that the formula ϕ holds for all paths starting from the current state and $\mathbf{E}\phi$ means that there is a path starting from the current state that satisfies ϕ. $\mathbf{G}\phi$ means all future states satisfy ϕ; $\mathbf{X}\phi$ means the next state of a path satisfies ϕ; $\mathbf{F}\phi$ means that ϕ holds in some future state; and $\phi\mathbf{U}\psi$ means that, if a future state (along a path) satisfies ψ, then all states from the current one to that ψ-satisfying state satisfy ϕ. Formulas combine the path quantifies and the temporal operations, e.g., $\mathbf{AG}\phi$ holds if ϕ holds on all states in all paths originating from the current state. This is also denoted as $\mathbf{A}[]\phi$ in Uppaal [1].

3.2 Our Uppaal model

In this section, we overview our Uppaal model of OLSR protocol; more details follow in Section 3.3.

We model OLSR in Uppaal as a parallel composition between node processes. Every process is a further parallel composition of two timed automata, `Queue` and `OLSR`, each having its own data structures. The `Queue` automaton has been chosen to model incoming messages from other nodes. In other words, it models the input buffer of a node: the received messages are buffered and then, in turn are sent to the `OLSR` automaton for processing. The `OLSR` automaton models the main OLSR process. It has local data structures to model the routing table and the broadcasting of control messages at particular times. Upon receiving a message by a node, its routing table is updated according to the information in the received message. Routing tables provide all the information required for route establishment and packet delivery. Every routing table `rt` is an array of entries, one entry for each node. An entry is modelled by the data type `rtentry`:

```
typedef struct {
  IP dip; //destination address
  int hops; //number of hops to the destination
  IP nhopip; //next hop along the path to the destination
  SQN dsn; //last sequence number of TC originator
 } rtentry;
```

Here, IP denotes a data type for all addresses and SQN a data type for sequence numbers. OLSR uses sequence numbers to check whether received messages are new or they have been processed before. In our model, integers are used to define these types. Every message is modelled by the data type MSG:

```
 typedef struct {
   MSGTYPE msgtype; //messages type:  PACKET, HELLO, TC
   IP oip; //the originator address of the message
   IP dip; //destination address of the message
   bool onehop[N]; //information about two-hop neighbours
   int ttltc; //max number of hops a message is forwarded
   int hops; //the distance to the receiver
   IP sip; //the sender address of a message
   SQN osn; //message sequence number showing freshness
   bool mpr[N]; //information about MPRs or MPR selectors
  } MSG;
```

Here, N is a constant denoting the number of nodes. msgtype shows type of the message flooded in the network and can have values PACKET, HELLO, or TC, oip is the address of the node who generates the message, dip is the destination address of the message, and it is used only for the PACKET, onehop is a boolean array of size N and is embedded in the HELLO message to give the information about two-hop neighbours of the receiver node, ttltc is an integer denoting the number of nodes in the network which represents how many hops a TC message can be forwarded, whenever a TC is forwarded, this value reduces by 1 , hops is an integer indicating the distance from the originator of the message to the receiver node, whenever a TC is forwarded, this value is increased by 1, sip is the address of the node which forwards a message and is used when generating a TC message, osn is the message sequence number shows the freshness of the message. The originator node assigns this identification number to each TC message, and boolean array of size N mpr represents the MPRs of the originator in HELLO messages and MPR selectors of the originator in TC messages.

Communication between two nodes is modelled by the isconnected[i][j] predicate as following:

```
bool isconnected(IP i, IP j){
  return(topology[i][j]==1);
  }
```

Here, topology is a two-dimensional boolean array of size N characterising the current configuration of the network. We do not model mobility nor failure of nodes in this paper and thus, topology models in fact the static structure of one network. The predicate holds only when nodes i and j are able to communicate. Communication between these two nodes is feasible if and only if they are in transmission range of each other.

Communication between nodes happens via channels. The *Broadcast* channel `htc[ip]` models the propagation of HELLO and TC messages where a message can be received by all directly connected nodes. Each node has a broadcast channel, and every node in the range may synchronise on this channel. We also use the *unicast* channel `packet[i][j]` to model the unicast sending of a data packet from `i` to `j`; this packet is generated by the user layer. Our model includes one channel for each pair of nodes and they are only enabled if they are directly connected.

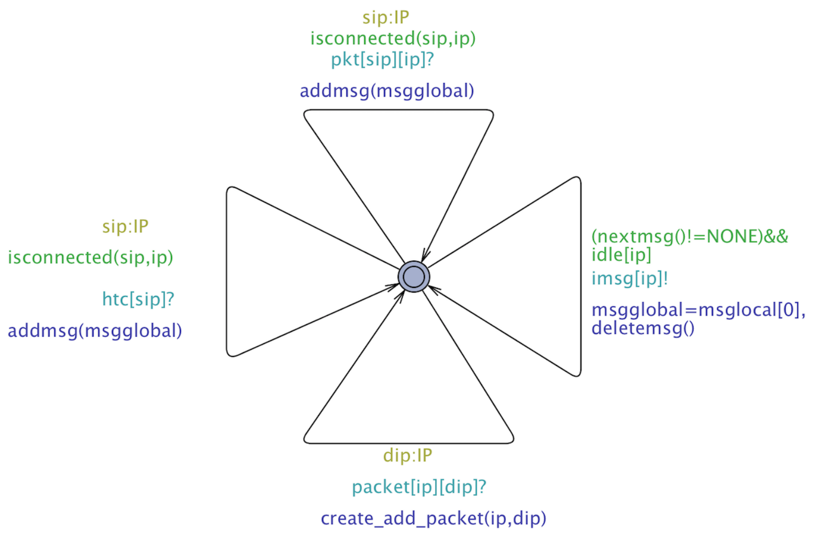

Fig. 2: `Queue` automaton.

The `Queue` automaton of a node `ip` is represented in Fig. 2. Each directly connected node receives messages from other nodes and stores them to its queue using the `addmsg` function:

```
void addmsg(MSG msg){
  msglocal[nodebuffersize[ip]]=msg;
  nodebuffersize[ip]++;
}
```

Here `msglocal` is a local array of size `QLength` (equals to 30) with elements of type `MSG` representing stored messages in the `Queue` and `nodebuffersize` is a global array of size N with integer elements. Global array `nodebuffersize` models the number of stored messages in `Queue`. Our model ensures that messages from those nodes which are in the transmission range are received. `Queue` automaton consists of some other functions such as `deletemsg`, `nextmsg` and `create_add_packet`, respectively to delete a message, return the type of the next message and to create a data packet. These functions are as following:

```
void deletemsg(){
  MSG empty_msg;
```

```
   for(i:  int[1,QLength-1]){
   msglocal[i-1]=msglocal[i];
 }
   msglocal[QLength-1]=empty_msg;
   nodebuffersize[ip]--;
 }}
MSGTYPE nextmsg(){
   return msglocal[0].msgtype;
 }
void create_add_packet(IP oip, IP dip){
   MSG msg;
   msg.msgtype= PACKET;
   msg.oip=oip;
   msg.dip=dip;
   addmsg(msg);
 }
```

We model the deletion of a message when `Queue` transfers the last message of its local array `msglocal`, i.e., `msglocal[0]`, to `OLSR` for processing, returning of the type of the next message to know about the type of `msglocal[0]` which is transferred to `OLSR`, and creation of a packet to be used in our experiments; checking packet delivery property.

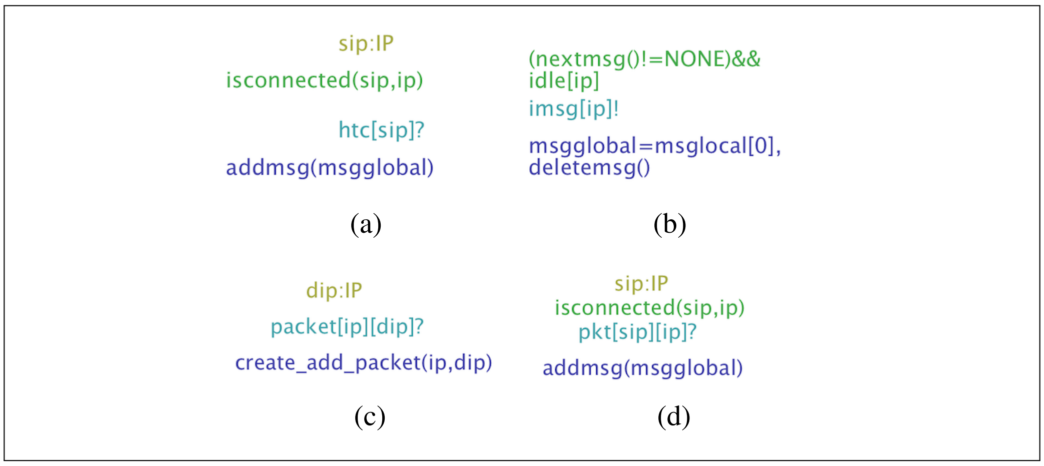

Fig. 3: `Queue` automaton transitions with selections, guards, synchronisation channel, and updates.

Fig. 3 depicts four transitions of `Queue` automaton with selections, guards, synchronisation channels and updates. In this figure, transition (a) is enabled when receiving a HELLO or TC message. transition (b) is taken when transferring relevant data from `Queue` to `OLSR` for processing, transition (c) is enabled when creating a PACKET and is used for our experiments, transition (d) is activated when receiving a PACKET from another node. This transition is also used in our experiments.

Transition (a) has selection `sip:IP` representing sender node of the messages as `sip` with type IP, guard `isconnected(sip, ip)` which shows whether or not nodes `sip` and the receiver are in transmission range of each other, channel `htc[sip]?` is the synchronisation channel where `Queue` as the receiver of the message adds `msgglobal` to its local array `msglocal` using `addmsg` as the update on this transition.

Transition (b) has a guard, a synchronisation channel and an update. The guards show if the type of the last message in `Queue` is HELLO, TC or PACKET and `OLSR` is not busy with processing other messages, `Queue` transfers the message to the `OLSR` using `imsg[ip]!`, assigns this last message into a global variable and at last deletes the message from its local array `msglocal`.

Transition (c), the one used for our experiments, has only a selection, a synchronisation channel and an update. Selection `dip:IP` indicates destination node of the packet as `dip` with type IP, synchronisation channel `packet[ip][dip]?` provides the possibility to create a PACKET applying `create_add_packet` function as the update.

Transition (d) has selection `sip:IP` showing sender node of the messages as `sip` with type IP, guard `isconnected(sip, ip)` which shows whether or not nodes `sip` and the receiver are in transmission range of each other, channel `pkt[sip][ip]?` is the synchronisation channel where `Queue` as the receiver of the message adds `msgglobal` to its local array using `addmsg` as the update on this transition.

The `OLSR` automaton modelling the message-handling protocol is more complicated. This automaton depicted in Fig. 4 has 12 locations and 27 transitions precisely modelling broadcasting and handling of the different types of messages, i.e., HELLO, TC and PACKET. `OLSR` is busy while sending messages, and can accept a new message from `Queue` only once it has completely finished handling a message. Whenever it is not processing a message, i.e., the boolean array `idle` of size N is equal to 1, and there are messages stored in `Queue`, `Queue` and `OLSR` synchronise on the urgent channel `imsg[ip]` meaning no delay must happen when transferring the relevant message data from `Queue` to `OLSR`. `OLSR` copies `msgglobal` to the local variable `msglocal` with type `MSG` and becomes not `idle`.

We use an urgent channel here to prevent the expiration of messages. In addition, we define two other channels called `tau[ip]` and `urg[ip]` for each `OLSR`: this means that we assign a higher priority to internal transitions, in order to reduce the state space. With this, we cannot check liveness properties, but for this paper it is not a limitation. All properties we are interested in can be represented as safety properties.

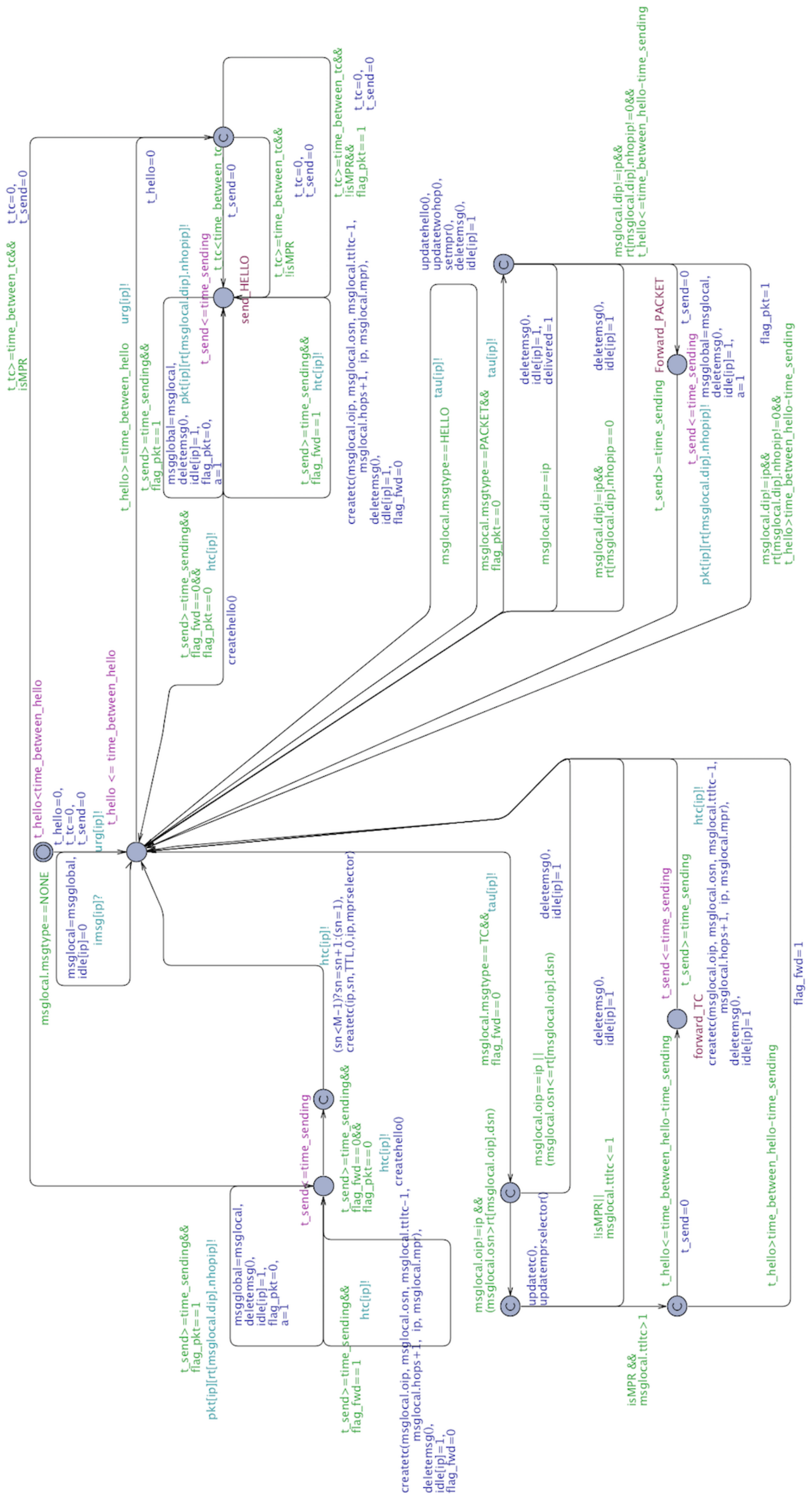

Fig. 4: `OLSR` automaton.

To model rigorous timing behaviour, we define 3 different clocks for every OLSR: t_hello and t_tc to model on-time broadcasting HELLO and TC messages, and t_send to model time consumed for sending messages. According to the specification of OLSR [4], each node broadcasts HELLO messages every 1500 milliseconds and the most time consuming activity, i.e., communication between nodes, can take up to 500 milliseconds. In our experiments, we assume always the maximum sending time, i.e., every message sending takes 500 milliseconds.

In order to be able to model the start-up working of nodes, we define the constant time_between_hello, that equals 1500; the starting points of nodes are varied between $[0, \texttt{time\_between\_hello})$. Different starting points provide means for modelling a realistic specification where nodes can start broadcasting at different times. As soon as each node starts working, we reset all clock variables to 0. Every time t_hello reaches 1500 milliseconds, we reset t_hello and t_send to 0 before transmission, and then we use an intermediate location which has the effect of selecting a delay of 500 milliseconds to model sending time. The HELLO message is created using the createhello function and copied to the global variable msgglobal as follows:

```
void createhello(){
  MSG msg;
  msg.msgtype= HELLO;
  msg.oip= ip;
  for(i:int[1,N-1]){
    if (rt[i].hops==1){ msg.onehop[i] = 1; }
    else {
      msg.onehop[i]= 0;
      if (rt[i].hops==2){ msg.mpr[rt[i].nhopip]=1;}
    }
  }
  msgglobal= msg;
}
```

OLSR assigns HELLO to the type of the messages and its ip to the message oip. If the hops of any entry in routing table, i.e., rt[i].hops, equals to 1, OLSR assigns 1 to the msgonehop[i]. Also, if rt[i].hops equals to 2, OLSR assigns 1 to the msgmpr of the next node for that entry. With this, OLSR is able to find its MPRs. The HELLO messages are then broadcasted. Connected nodes receive the HELLO message, update their routing tables for the originator of the message, learn about their two-hop neighbours and select their MPRs and MPR selectors using functions called updatehello, updatetwohop and setmpr functions, respectively.

```
void updatehello(){
    rt[msglocal.oip].dip=msglocal.oip;
    rt[msglocal.oip].hops= 1;
    rt[msglocal.oip].nhopip=msglocal.oip;
    rt[msglocal.oip].dsn=rt[msglocal.oip].dsn;
}
```

`OLSR` updates the routing table for the message originator, i.e., `msglocal.oip`. It assigns the message originator address to the destination address in the routing table. It means that `rt[msglocal.oip].dip` is updated by assigning `msglocal.oip` into it. `hops` is assigned to 1, the next node is the originator node: `rt[msglocal.oip].nhopip=msglocal.oip`, and since HELLO messages do not have sequence numbers, `rt[msglocal.oip].dsn` remains unchanged.

```
void updatetwohop(){
    for(i:int[1,N-1]){
      if(msglocal.onehop[i]==1 && i!=ip &&
      rt[i].hops!=1){
        rt[i].dip=i;
        rt[i].hops= 2;
        rt[i].nhopip= msglocal.oip;
     }
   }
 }
```

Update of two-hop neighbours happens in this stage based on the information about the one-hop neighbours of the originator, i.e., `msglocal.onehop` . If any elements of this array equals to 1, and if that element's address is not equal to the address of the node who is processing the message and also if the `hops` of the routing table for that element has not been updated before while using `updatehello` function, the routing table for that element is updated as following: (1) the address of that element is assigned into the destination address, i.e., `rt[i].dip`, (2) `rt[i].hops` is assigned to 2, and (3) the message address originator is assigned to `rt[i].nhopip` since the message originator is the next node to two-hop neighbours.

```
void setmpr(){
    if (msglocal.mpr[ip] == 1){
      isMPR=1;
      mprselector[msglocal.oip]=1;
   }
}
```

Here, `mprselector` is a local boolean array of size N which indicates the MPR selectors of every node. This function models the process of selecting MPRs and also finding MPR selectors in a way that if any element of the `mpr` array of the received message is equal to 1, the boolean array isMPR is changed to 1, and mprselector array is updated for the message originator which means that mprselector[msglocal.oip]=1.

At the next step, after MPR nodes are selected (i.e., local boolean variable isMPR equals 1), and MPR selectors are determined, every MPR node broadcasts TC messages to the connected nodes every 4500 milliseconds. For this, we define the constant time_between_tc, that equals 4500. When t_tc reaches 4500, t_tc and t_send are reset to 0 before transmission and again we use another intermediate location to let OLSR select a delay of 500 milliseconds. Then, a TC message is generated by createtc function and is broadcasted to other nodes:

```
 void createtc(IP oip, SQN osn, int ttltc, int hops, IP
sip, bool mprselector[N]){
   MSG msg;
   msg.msgtype= TC;
   msg.oip= oip;
   msg.osn= osn;
   msg.ttltc= ttltc;
   msg.hops= hops;
   msg.sip= sip;
   msg.mpr = mprselector;
   msgglobal= msg;
  }
```

While receiving a TC message from Queue, if the message is considered for processing, the routing table is updated for the TC generator and its MPR selectors, using updatetc and updatemprselector functions, respectively.

```
 void updatetc(){
   if(rt[msglocal.oip].hops==1||
   rt[msglocal.oip].hops==2){
      rt[msglocal.oip].dsn=msglocal.osn;}
   else{
   if(rt[msglocal.oip].hops!=1&&
   rt[msglocal.oip].hops!=2){
      rt[msglocal.oip].dip=msglocal.oip;
      rt[msglocal.oip].hops= msglocal.hops+1;
      rt[msglocal.oip].nhopip= msglocal.sip;
      rt[msglocal.oip].dsn=msglocal.osn;
     }
   }
}
```

Here, if the routing table has been updated for the originator of the message while receiving a HELLO, the routing table is updated only for the sequence number of the originator. It means that `rt[msglocal.oip].dsn` is substituted by `msglocal.osn`. If the routing table has not been updated for the originator of the messages, the routing table is updated as following:

- message originator address is assigned to the `rt[msglocal.oip].dip`
- `rt[msglocal.oip].hops` is substituted by `msglocal.hops+1`
- the sender address of the message is assigned as the next hop along the path to `rt[msglocal.oip].dip`
- the message sequence number is assigned to `rt[msglocal.oip].dsn`

Function `updatemprselector` is used to update the information about the MPR selectors of each MPR. If an element of `msglocal.mpr` equals to 1, the routing table for that element has not been updated before and that element address is not equal to the address of the receiver node, the routing table for such element is updated as following:

- the address of the element, i.e., i, is assigned to `rt[msglocal.oip].dip`
- `rt[msglocal.oip].hops` is substituted by `msglocal.hops+2`
- the sender address of the message is assigned as the next hop along the path to `rt[i].dip`
- routing table is not updated for `dsn`. We update `rt[msglocal.oip].dsn` only in `updatetc` function when the routing table is updated for the originator of a TC message.

```
void updatemprselector(){
 for(i:int[1,N-1]){
  if (msglocal.mpr[i]==1 && rt[i].hops!=1 &&
  rt[i].hops!=2 && i!=ip){
   rt[i].dip=i;
   rt[i].hops=msglocal.hops+2;
   rt[i].nhopip=msglocal.sip;
   rt[i].dsn=rt[i].dsn;
   }
  }
}
```

If the receiver node is an MPR and the TC message is considered for forwarding, the message is forwarded. Forwarding messages also takes time in our model 500 milliseconds. We should add here that OLSR might have to broadcast different messages at the same time. As an example, at some point a HELLO, a TC and maybe a TC to be forwarded are supposed to be broadcasted; the sending time, i.e., 500 milliseconds, is counted only once and these messages are broadcasted simultaneously. We considered this behaviour in our model using committed locations and boolean variables. For instance, `flag_fwd` and `flag_pkt` are examples of such boolean variables. If the value of these variables equals 1, it indicates that there is a message in the node, considered for forwarding and waiting to be forwarded together with node's control messages.

We also model the behaviour of OLSR in case of receiving and forwarding a packet. Due to the proactive nature of the OLSR, if one node receives a packet, it must have the information about the destination of that packet in its routing table, to forward the packet to the next node along the path to the destination node. In case the node has to broadcast its own control messages, the packet waits in the node to be forwarded together with node's control messages; it means that `OLSR` changes the value of `flag_pkt` to 1, showing that a packet is waiting to be forwarded.

3.3 Functioning of the `OLSR` automaton

The initial state of `OLSR` has the invariant `t_hello<time_between_hello` as shown in Fig. 4. Upon taking the `urg[ip]` channel transition from the initial state, all the clock variables are assigned to 0, to support the realistic behaviour of nodes mentioned in Section 3.2, and `OLSR` moves to the central state which has the invariant `t_hello<= time_between_hello`. Every time the guard on the next `urg[ip]` channel transition outgoing from the central location is satisfied, i.e., `t_hello` 1500, we reset `t_hello` to 0 before transmission, and then we use an intermediate committed location. There are four transitions outgoing from this committed location (the committed location in the up right corner side) with the following guards:

$$\texttt{t\_tc} < \texttt{time\_between\_tc} \tag{1}$$

$$\texttt{t\_tc} >= \texttt{time\_between\_tc} \;\&\&\; \texttt{!isMPR} \tag{2}$$

$$\texttt{t\_tc} >= \texttt{time\_between\_tc} \;\&\&\; \texttt{!isMPR} \;\&\&\; \texttt{flag\_pkt} == \texttt{1} \tag{3}$$

$$\texttt{t\_tc} >= \texttt{time\_between\_tc} \;\&\&\; \texttt{isMPR} \tag{4}$$

If any of guards (1), (2) or (3) is satisfied, `OLSR` goes to the next intermediate location with invariant `t_send<= time_sending` (`time_sending` 500) and does the updates. If the transition with guard (1) is taken, `t_send` is assigned to 0, and if transitions with guards (2) or (3) are taken, both `t_send` and `t_tc` are assigned to 0. In this location (`send_HELLO`), there are three self-transitions with the following guards:

$$\texttt{t\_send} >= \texttt{time\_sending} \;\&\&\; \texttt{flag\_pkt} == \texttt{1} \tag{5}$$

$$\texttt{t\_send} >= \texttt{time\_sending}\ \&\&\ \texttt{flag\_fwd} == 1 \tag{6}$$

$$\texttt{t\_send} >= \texttt{time\_sending}\ \&\&\ \texttt{flag\_fwd} == 0\ \&\&\ \texttt{flag\_pkt} == 0 \tag{7}$$

The transition with guard (5) is taken if `t_send` reaches 500 and there is a packet to be forwarded: in this case, `OLSR` copies `msglocal` (which is a packet) to the global variable `msgglobal` and sends the packet to the next node along the path of the destination, deletes the message using the `deletemsg` function as shown below, assigns `idle` to 1, and `flag_pkt` to 0, changes `a` to 1 showing that the packet is forwarded by the node. Then, it goes back to the same location, (`send_HELLO`).

```
void deletemsg(){
  msglocal.msgtype=NONE;
}
```

The transition with guard (6) is taken if `t_send` reaches 500 and there is a TC to be forwarded. In this case, the following happens: `OLSR`

- reduces the value of the `ttltc` by 1
- increases the `hops` variable of the message by 1
- assigns its own address as the sender of the message
- copies the `msglocal` which is a TC to the global variable `msgglobal`
- forwards the TC
- deletes the message by `deletemsg` function
- assigns `idle` to 1 and `flag_fwd` to 0
- and finally goes to the same location, (`send_HELLO`).

Transition with guard (7) is taken if `t_send` reaches 500 and there is no TC or packet to be forwarded; in this situation `OLSR` creates the HELLO message and broadcasts it. Then, it moves to the central location.

In case of taking the transition with (4), `t_send` and `t_tc` are assigned to 0 and `OLSR` goes to the next delay location (the location in the up left corner side). In this location, three self-transitions can be taken with respect to the guards on those transitions. These guards are as following:

$$\texttt{t\_send} >= \texttt{time\_sending}\ \&\&\ \texttt{flag\_pkt} == 1 \tag{8}$$

$$\texttt{t\_send} >= \texttt{time\_sending}\ \&\&\ \texttt{flag\_fwd} == 1 \tag{9}$$

$$\texttt{t\_send} >= \texttt{time\_sending}\ \&\&\ \texttt{flag\_fwd} == 0\ \&\&\ \texttt{flag\_pkt} == 0 \tag{10}$$

The description of these guards and transitions is similar to those of (5), (6) and (7). The only difference is on the third guard; when transition with guard t_send>=time_sending && flag_fwd==0 && flag_pkt==0 is taken, OLSR does the updates in the transition and moves to the next committed location. Then immediately, it creates and broadcasts its TC message and goes back to the central location.

Upon receiving a HELLO message from Queue, i.e., transition with guard msglocal.msgtype==HELLO, each node updates its routing table for one-hop and two-hop neighbours, selects its MPRs, and finally deletes the HELLO message and changes the value of idle to 1.

While receiving a TC from Queue, i.e., msglocal.msgtype==TC && flag_fwd==0, the location is changed to the committed location where two transitions with guards have to be taken. We should mention that flag_fwd==0 indicates that no TC message is waiting in OLSR to be forwarded. We use this flag to prevent taking this transition several times when a TC to be forwarded is still in OLSR. These guards are as following:

$$\texttt{msglocal.oip} == \texttt{ip} \;||\; (\texttt{msglocal.osn} <= \texttt{rt}[\texttt{msglocal.oip}].\texttt{dsn}) \quad (11)$$

$$\texttt{msglocal.oip}! = \texttt{ip} \;\&\&\; (\texttt{msglocal.osn} > \texttt{rt}[\texttt{msglocal.oip}].\texttt{dsn}) \quad (12)$$

In guard (11), we check if the receiving node is the originator of the message; or, if the sequence number of the message is smaller than or equal to the sequence number of the routing table for the message originator, we delete the message and change idle to 1. In guard (12), we check if the receiving node is not the originator of the message, and the sequence number of the message is greater than the sequence number of the routing table for the message originator, the message is considered for processing. The routing table for the message originator and MPR selectors of the message originator are updated and in parallel, OLSR reaches the next committed location to decide if the message needs to be forwarded or not.

If the receiver node is not an MPR or the message cannot be forwarded anymore, i.e., !isMPR || msglocal.ttltc<=1, the TC message is deleted and idle is assigned to 1. If the receiving node is an MPR and the message can be still forwarded, i.e., !isMPR && msglocal.ttltc>1, the message is considered for forwarding (next committed location). While the TC is going to be forwarded, there might be the possibility that OLSR has to broadcast its own control messages when t_hello>time_between_hello-time_sending. In this case, flag_fwd is set to 1, OLSR goes back to the central location to broadcast its own control messages and the TC to be forwarded waits to be broadcasted together with the node's control messages. However, in case the guard t_hello<=time_between_hello-time_sending is satisfied, meaning that the node does not have to broadcast its own control messages, the TC message can be forwarded considering the intermediate delay location (Forward_TC). After forwarding, the TC is deleted and OLSR becomes idle.

When OLSR receives a PACKET from Queue, i.e., transition with guard msglocal.msgtype==PACKET && flag_pkt==0, the location is changed to the committed location where four transitions with some guards have to be taken. We should mention here that flag_pkt==0 indicates that no PACKET is waiting in OLSR to be forwarded. We use this flag to prevent taking this transition several times when a PACKET to be forwarded is still in OLSR. These guards are as following:

$$\texttt{msglocal.oip} == \texttt{ip} \tag{13}$$

$$\texttt{msglocal.oip}! = \texttt{ip}\ \&\&\ \texttt{rt}[\texttt{msglocal.dip}].\texttt{nhopip} == 0 \tag{14}$$

$$\begin{aligned}\texttt{msglocal.dip}! = \texttt{ip}\ \&\&\ \texttt{rt}[\texttt{msglocal.dip}].\texttt{nhopip}! = 0\ \&\&\\ \texttt{t\_hello} <= \texttt{time\_between\_hello} - \texttt{time\_sending}\end{aligned} \tag{15}$$

$$\begin{aligned}\texttt{msglocal.dip}! = \texttt{ip}\ \&\&\ \texttt{rt}[\texttt{msglocal.dip}].\texttt{nhopip}! = 0\ \&\&\\ \texttt{t\_hello} > \texttt{time\_between\_hello} - \texttt{time\_sending}\end{aligned} \tag{16}$$

Guard (13) represents if the receiving node is the destination node, the PACKET is deleted from OLSR, idle value is assigned to 1 and delivered which shows if the PACKET is received by the destination is assigned to 1. Guard (14) indicates if the receiving node is not the destination node and the routing table for the destination node has not been not updated, the packet is dropped and idle changes to 1. Guard (15) shows if the receiving node is not the destination node, the information about the destination node is available, and the receiver does not have to broadcast its own control messages, the PACKET is forwarded along the path to the destination node considering the sending time applying the intermediate delay location (Forward_PACKET). Guard (16) indicates that if the receiving node is not the destination node, the information about the destination node is available, and the node has to broadcast its own control messages, as a consequence, it assigns flag_pkt to 1 which presents that the PACKET must be forwarded with the node's control messages simultaneously.

4 Experiments

The main purpose of using Uppaal is to verify our OLSR model w.r.t. the requirement specification. Our automated analysis of OLSR considers 3 properties that relate to route establishment for all topologies up to 5 nodes, packet delivery in different network topologies with at most 5 nodes, and a scenario which reports on finding non-optimal routes with 7 nodes. The latter was discovered during the creation of the formal specification in Uppaal.

Due to proactive nature of OLSR, our Uppaal model has become quite complex with many states; adding one more node to the system makes the verification part longer and more complicated (in some cases, we could even not verify properties for 5 nodes). As a consequence, we applied different techniques to avoid state space explosion and to minimise our system model inspired from Uppaal literature [5, 16, 17].

We added priority on channels to overcome the state space problem. For the first two experiments, we give the highest priority to channels of node `a1` and the lowest priority to channels of node `a5`. The priority of other node channels are in between these two. This priority assigning contributes to overcoming the state space problem and is not a limitation for this paper because we check the first two properties for all topologies up to 5 nodes: this means that `a1`, the starting node with highest priority, will be in every location in different network topologies. For the third property we also assign the highest priority to node `a1` and the lowest priority to node `a7`.

The experiments use the following set up: 3.2 GHz Intel Core i5, with 8 GB memory, running the Mac OS X 10.9.5 operating system. For all experiments we use Uppaal 4.0.13.

4.1 Properties

We now detail the properties that we have verified. In our experiment, we assume that the originator is always `a1`, denoted by `OIP1`, and the destination is always `a5`, denoted by `DIP1`. For our experiments, we defined another automaton called `Tester` which injects a data packet to `OIP1` to be delivered at the destination `DIP1`. This automaton is illustrated in Fig. 5. The statements written under the locations are the invariants of the two specific locations, while the guards of the transitions are written above them, together with the potential update and synchronisation channel. The `Tester` has a local clock named `clk`, 3 locations and 2 transitions with clock guards. When `clk` reaches `3*(time_between_tc)`, `Tester` synchronises with `Queue` of `OIP1`, resets `clk` to 0, and moves to location `test`. In parallel, the `Queue` of `OIP1` creates a data packet to be sent to the destination `DIP1`. The `Tester` waits in `test` location until `clk` reaches `5*(time_between_tc)` to move to location `delivery`. We model `3*(time_between_tc)` and `5*(time_between_tc)` to denote some arbitrary time periods, so that we assume required TC messages are received by nodes to update their routing tables.

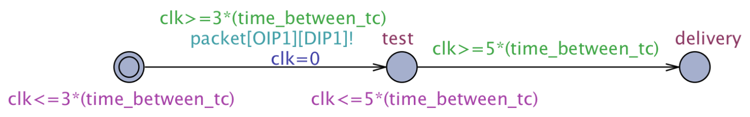

Fig. 5: `Tester` automaton.

The first property states that, after `Tester` and `Queue` of `OIP1` are synchronised (`Tester` in location `test`), a route from the originator to the destination has been found. This safety property using the Uppaal syntax is expressed as:

$$\texttt{A[ ] (Tester.test imply node(OIP1).rt[DIP1].nhopip!} = 0) \qquad (17)$$

The CTL formula `A[]`ϕ is satisfied if ϕ holds on all states along all paths. The variable `node(OIP1).rt` represents the routing table of the originator node `OIP1`, and `node(OIP1).rt[DIP1].nhopip` expresses the next hop for the destination `DIP1`. All required HELLO and TC messages are sent if and only if `Tester` is in location `test`; the originator `OIP1` has a route to node `DIP1` if and only if `node(OIP1).rt[DIP1].nhopip` is not equal to 0.

The second property is that if a packet is injected to the system via the user, it is delivered to the destination `DIP1`. In Uppaal this can be expressed as:

$$\mathtt{A[\,]}\,(\mathtt{Tester.delivery\ imply\ node(DIP1).delivered!} = \mathtt{0}) \tag{18}$$

The variable `node(DIP1).delivered` shows whether the injected data packet is received by the destination `DIP1`. This property is satisfied if `Tester` is in location `delivery` and for all paths, `DIP1` has always updated the boolean value `delivered`, i.e., `node(DIP1).delivered` is not 0.

The third property states that, after broadcasting, forwarding and processing TC messages, OLSR would guarantee an optimal route w.r.t. hop count. We investigate this property for a network topology with 7 nodes as shown in Fig. 2 . This property is expressed as:

$$\begin{aligned}\mathtt{A[\,]}\,((\mathtt{Tester.test}\ \&\&\ \mathtt{node(OIP1).a!} = \mathtt{0})\ \mathtt{imply}\\ \mathtt{node(OIP1).rt[DIP1].hops} == \mathtt{3})\end{aligned} \tag{19}$$

Here, variable `node(OIP1).a!=0` indicates whether or not `OIP1` has sent its packet to the next node along the path to the destination `DIP1`, and variable `node(OIP1).rt[DIP1].hops` shows the number of hops from the originator `OIP1` to the destination `DIP1` which must be equal to 3. We use 7 nodes network for verifying optimal route finding property. Although OLSR is able to find optimal routes in small networks by updating routing tables while receiving HELLO messages, we have uncovered that a node might find non-optimal routes in larger networks.

4.2 Results

To analyse and verify our model, we evaluate it in all network topologies up to 5 nodes. Property (17) was satisfied for all these networks; when the `Tester` is in location `test`, node `OIP1` has established a route to node `DIP1`. This property confirms the propagation of HELLO and TC messages and also the correctness of the MPR selection mechanism. Hence, node `OIP1` is ready to send data packets to node `DIP1`.

Property (18) is stricter than property (17). It models that all nodes have the information about all other nodes in the network, to deliver their data packets. In theory, the originator node `OIP1` could have a routing table entry for the destination node `DIP1`, stating that it should send a packet to its immediate next neighbour along the path to the destination `DIP1`; the next node itself might have no information about the destination, so all packets for the destination `DIP1` stemming from the originator `OIP1` would be lost. However, property (18) is also checked by the Uppaal verifier: this means that all nodes have updated their routing tables for all other nodes in the network. Therefore, they are able to deliver data packets to the arbitrary destination node `DIP1`.

Interestingly, property (19) related to the optimal route finding for 7 nodes was not satisfied. This indicates that OLSR is not always able to find optimal routes. We illustrate this phenomenon with the example found by Uppaal, with the following steps shown in Table. 2. In this example, `Tester` synchronises with the `Queue` of `OIP1` (`Tester` is in location `test`), and `OIP1` has sent the created data packet to the next node along the path to the destination `DIP1`. Based on the proactive nature of OLSR, all required information is provided in routing tables; the Uppaal simulator shows that node `OIP1` has sent its data packet via node `a2` to the destination, which is not the optimal route! The problem is that, while node `a5` is broadcasting its TC to nodes `a4` and `a6` (Table 2: Step 1), node `a4` might forward TC5 earlier. Then this TC message would be forwarded subsequently earlier via nodes `a3` and `a2` (Table 2: Step 2). As a consequence, node `OIP1` updates its routing table for node `a5` (Table 2: Step 3) and when it receives another TC5 via node `a7`, it has already updated its table for this node. Since the sequence number of the TC message is the same, it drops the TC5 arrived via shorter hops (Table 2: Step 4).

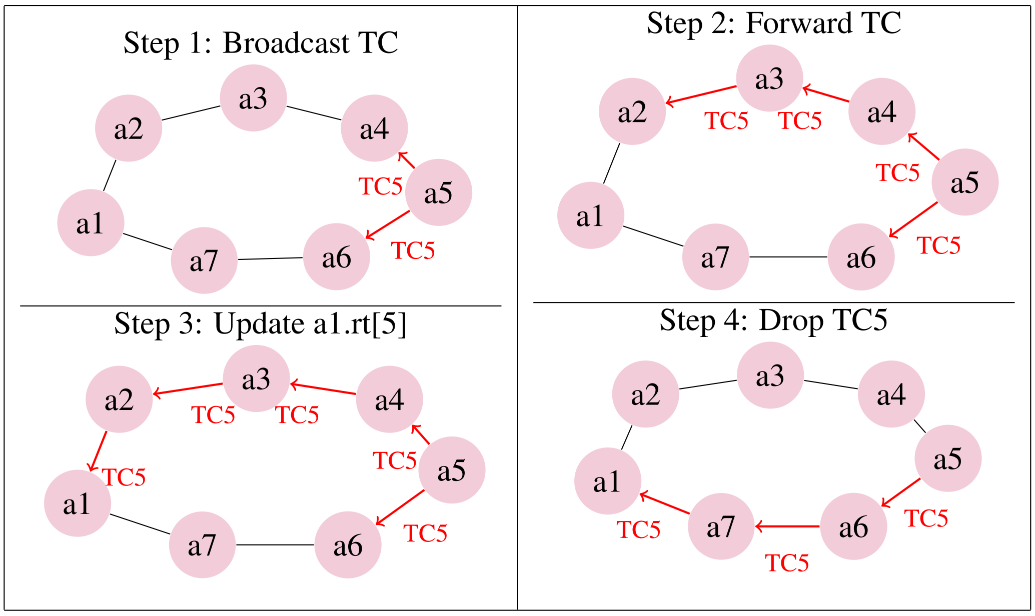

Table 2: Non-optimal Route in a 7 nodes topology

"if there exists a tuple in the duplicate set, where:
D_addr == Originator Address,
AND
D_seq_num == Message Sequence Number
then the message has already been completely processed and MUST not be processed again." [RFC3626, page 16]

Uppaal works correctly; it is the specification of OLSR that prescribes this functions, as shown above. In our model, `D_addr` is modelled by `dip`, and `D_seq_num` by `dsn`. So, the generated data packet is delivered via the longer route. One solution can be as follows. When processing a TC messages, if the sequence number of the messages is equal to the last sequence number of the TC originator in the routing table, the number of hops should be checked; if hops of the message is bigger than the one in the routing table, the message must be dropped. But if it is smaller, the message must be processed again.

5 Related Work

While modelling and verifying protocols is not a new research topic, attempts to verify routing protocols for WMNs are still rather new and remain a challenging task. Model checking techniques have been applied to analyse protocols for decades, but there are only a few papers that use these techniques in the context of mobile ad-hoc networks, e.g., [2]. In the area of WMNs, Uppaal has been used to model and analyse the routing protocols AODV and DYMO, see [8, 9, 11]. In the following, we overview some research related to the work in this paper. However, to the best of our knowledge, our study is the first aiming at a formal model of OLSR core functionality considering time variables.

Clausen et al. [4] specify the OLSR routing protocol used in mobile ad-hoc networks. This paper is the official description currently standardised by the IETF. Jacquet et al. [13] also provide a high-level description of OLSR describing the advantages of this protocol, when compared to the others.

Steele and Andel [19] provide a study of OLSR using the Spin model checker [12]. They designed a model of OLSR in which Linear Temporal Logic (LTL) is used to analyse the correct functionality of this protocol. They verified their system for correct route discovery, correct relay selection, and loop-freedom. Due to the state space explosion problem, their modelling is limited to four node networks.

Fehnker et al. [9] describe a formal and rigorous model of the AODV routing protocol in Uppaal; this is derived from a precise process-algebraic model that reflects a common and unambiguous interpretation of the RFC [18]. They model each node in the network as an automaton. They also describe some experiments for exploring AODV's behaviour in all network topologies up to 5 nodes. Although the two protocols AODV and OLSR behave differently, we use the same modelling techniques and experiments as for AODV, to make the comparison study of these two protocols feasible for our future work.

Kamali et al. [14] have used refinement techniques for modelling and analysing wireless sensor-actor networks. They prove that failed actor links can be temporarily replaced by communication via the sensor infrastructure, given some assumptions. They have used Event-B formalisation based on theorem proving and their proofs are carried out in the RODIN tool platform. There is a strong similarity between the nature of the distributed OLSR protocol and the nature of distributed sensor-based recovery. However, the tools employed for analysis in the two frameworks are different in nature (model checking vs. theorem proving) and hence the results are also different.

6 Conclusions and Outlook

The concrete result of this paper consists in providing a formal analysis for a distributed and proactive routing protocol named OLSR. Our analysis is performed based on the Uppaal model checker. The analysis shows that our Uppaal model is in accordance with the OLSR standard specification, but also points out a weakness of the protocol: in some cases, an optimal route for message delivery cannot be found. We also sketch a solution for the uncovered problem.

We see these results as the starting point for at least two directions of future research. First, our analysis is restricted to small networks (of 5 and 7 nodes), due to the nature of model checking. Wireless Mesh Networks draw their strength from employing potentially thousands of nodes (or more), hence, we need to extend our analysis to larger networks. This can be achieved by working with statistical model checking, where simulation concepts are combined with model checking to establish the statistical evidence of satisfying hypotheses. While this does not guarantee a correct result w.r.t the hypothesis, the probability of error can be made vanishingly small. Another approach suitable to deal with larger networks is that of theorem-proving, where, e.g., we can prove the required system properties as invariants for all systems (of all sizes) that verify certain assumptions. Theorem-proving is traditionally seen as difficult to carry out, but the advent of tools considerably eases the modelling: the needed proofs are automatically generated and even partly discharged, while the remaining proof obligations are dealt with interactively by the user of the theorem-proving tool.

Second, our model for the proactive, distributed OLSR can be generalised to distributed control. The latter is a concept with high relevance for systems where, e.g., self-repairing is important, as it can enable the independence of the system from central coordinators. Even maintaining proactively the optimal communication routes, as OLSR does, is instrumental in this. The applicability of distributed control to critical systems such as emergency response networks or smart electrical grids is very relevant, as these are complex systems, for which global solutions cannot be provided.

Acknowledgement We would like to thank Dr. Peter Höfner for his timely support with respect to Uppaal modelling and OLSR functioning.

References

[1] Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004. Revised Lectures. pp. 200–236. Springer Verlag (2004)

[2] Chiyangwa, S., Kwiatkowska, M.Z.: A timing analysis of AODV. In: FMOODS. Lecture Notes in Computer Science, vol. 3535, pp. 306–321. Springer (2005)

[3] Clarke, E.M., Emerson, E.A., Sifakis, J.: Model checking: Algorithmic verification and debugging. Commun. ACM 52(11), 74–84 (2009)

[4] Clausen, T., Jacquet, P.: Optimized link state routing protocol (OLSR). RFC 3626 (Experimental) (2003), `http://www.ietf.org/rfc/rfc3626`

[5] David, A., Håkansson, J., Larsen, K.G., Pettersson, P.: Model checking timed automata with priorities using DBM subtraction. In: 4th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS06). Lecture Notes in Computer Science, vol. 4202, pp. 128–142. Springer Berlin Heidelberg (2006)

[6] Emerson, E.A.: Temporal and modal logic. In: Handbook of Theoretical Computer Science (vol. B): Formal Models and Semantics, pp. 995–1072. MIT (1995)

[7] Fehnker, A., van Glabbeek, R.J., Höfner, P., McIver, A., Portmann, M., Tan, W.L.: A process algebra for wireless mesh networks. In: European Symposium on Programming (ESOP'12). Lecture Notes in Computer Science, vol. 7211, pp. 295–315. Springer (2012)

[8] Fehnker, A., van Glabbeek, R., Höfner, P., McIver, A., Portmann, M., Tan, W.L.: Modelling and analysis of AODV in UPPAAL. In: 1st International Workshop on Rigorous Protocol Engineering. pp. 1–6. Vancouver (2011)

[9] Fehnker, A., van Glabbeek, R., Höfner, P., McIver, A., Portmann, M., Tan, W.L.: Automated analysis of AODV using UPPAAL. In: 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2012). pp. 173–187. Springer, Tallinn, Estonia (2012)

[10] van Glabbeek, R., Höfner, P., Portmann, M., Tan, W.L.: Sequence numbers do not guarantee loop freedom —AODV can yield routing loops—. In: Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM'13). pp. 91–100. ACM (2013)

[11] Höfner, P., McIver, A.: Statistical model checking of wireless mesh routing protocols. In: 5th NASA Formal Methods Symposium (NFM 2013). vol. 7871, pp. 322–336. Springer, Moffett Field, CA, USA (2013)

[12] Holzmann, G.J.: The model checker spin. IEEE Trans. Softw. Eng. 23(5), 279–295 (1997)

[13] Jacquet, P., Mühlethaler, P., Clausen, T., Laouiti, A., Qayyum, A., Viennot, L.: Optimized Link State Routing Protocol for Ad Hoc Networks. In: Multi Topic Conference, 2001. IEEE INMIC 2001. pp. 62 – 68. IEEE (2001)

[14] Kamali, M., Laibinis, L., Petre, L., Sere, K.: Formal development of wireless sensor-actor networks. Science of Computer Programming 80, Part A(0), 25 – 49 (2014)

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

[16] Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: Compact data structures and state-space reduction for model-checking real-time systems. Real-Time Systems 25(2-3), 255–275 (2003)

[17] Larsen, K.G., Pettersson, P., Yi, W.: Model-checking for real-time systems. In: FCT. pp. 62–88 (1995)

[18] Perkins, C., Belding-Royer, E., Das, S.: Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561 (Experimental) (2003), `http://www.ietf.org/rfc/rfc3561`

[19] Steele, M.F., Andel, T.R.: Modeling the optimized link-state routing protocol for verification. In: SpringSim (TMS-DEVS). pp. 35:1–35:8. Society for Computer Simulation International (2012)

ISBN 978-952-12-3151-3
ISSN 1239-1891