Formal Verification DSR protocol simulation in Omnet++ using SPIN

Emir Kuanyshev
Nazarbayev University
Astana, Kazakhstan
emir.kuanyshev@nu.edu.kz

I. Introduction

The Network protocols are the established rules that determine how different nodes, such as mobile, laptop, and other devices, are connected. In other words, network protocols are the backbone of communication systems that correctly send, modify, and receive data. For example, routing protocols such as DSR ensure effectiveness using route caching and recovery from error algorithms in dynamic network topology, where nodes and connections can be permanently changed.

The algorithms of the network protocol and overall behavior can be tested via simulation. The simulation can be created using a framework such as OMNeT++, which can create a network topology and simulate the message traveling according to the rules. However, OMNeT++ does not ensure the correctness of the results as it heavily depends on the accuracy of the algorithm used. That is to say, we can not be sure we are getting the correct results even though the simulation was successfully started and finished.

Thus, the algorithm must be further verified to ensure that the models used in the simulation will simulate expected behavior and results. Simulation can be verified using formal verification tools such as SPIN/Promela. That is to say, SPIN, which can verify the correctness of the concurrent systems, can precisely verify the simulation of the network protocols' code from the OMNeT++. Thus, we can verify such properties as deadlock-freedom, liveliness and message delivery guarantees.

This paper shows how the network protocol's code from OMNeT++ can be translated into SPIN models. Additionally, it shows which properties of the Network Protocols SPIN can verify to ensure the correctness of the algorithms and simulation.

II. Network Topology

A. Network

The simulation involves the network forming a grid of nodes where each node is connected to the neighbors horizontally and vertically. For example, the network with a grid size equal to 2 is depicted in Figure 1. This network's topology was selected to easily scale the number of nodes as the network is automatically created using loops to form a grid of nodes using gridSize param. Additionally, Figure 2 shows how the small part of the grid looks with 400 nodes.

For a better understanding, you can watch the DSR protocol simulation in OMNeT++ with 16 nodes in the recorded video by following this link.

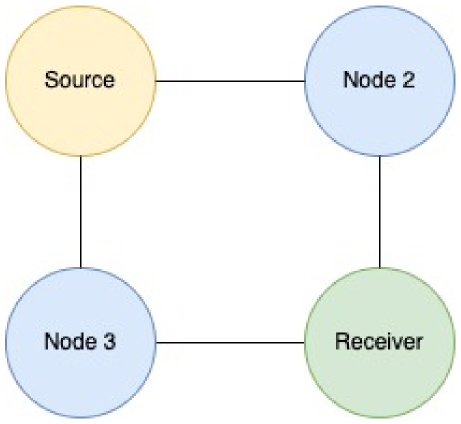

Fig. 1. Grid Topology.

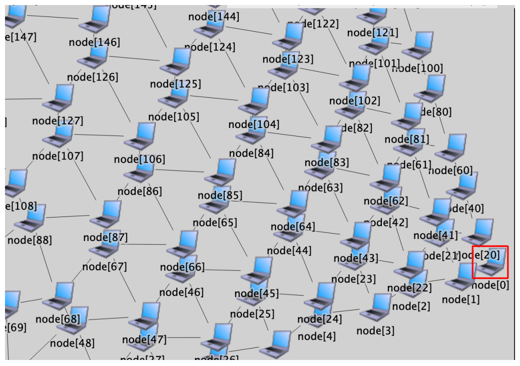

Fig. 2. Grid Topology.

B. Node

Each node consists of the following parameters:

- **isSource** - determines whether the node is the source
- **destination** - the destination node from the source.
- **numMessages** - number of messages that will be sent with **messageInterval** from the source to the destination. The numMessages is used to simulate how different messages will be delivered to the destination for better metrics measurement

- **cacheTTL** - the time limit for storing the cached route. Used for the DSR implementation.

C. Parameters

The network protocols were simulated with 400 nodes, the first being the source and the last with index 399 being the destination and receiver. Additionally, the cache TTL was equal to 5 seconds, messages were equal to 10 messages, and there was a 1-second interval between those 10 messages.

III. Network Protocols Implementation

The DSR protocol broadcasts the RREQ message to the connected neighbors by randomly selecting available gates. The list of available gates is selected and randomly shuffled to simulate the different message paths. This part of the code helps prevent duplicate paths in the simulations from becoming similar to real-world cases. The implementation also uses sequence numbers to avoid duplicated message paths. Additionally, every message is sent with a 0.1 delay to simulate the propagation time from one node to another. Another essential thing to add is that DSR uses route identification with RREQ and route caching using RREP. Thus, it helps us use cached routes, accelerating the messages' travel. Finally, ten messages were sent with a 1-second interval to measure end-to-end delay and routing overhead and how they change with each message.

In the next section, graphs provide the outputs from the simulation of the DSR Network Protocol in OMNeT++. Showing the simulation created approximately correct results. Thus, we can further investigate and compare the results with the outputs from the formal verification.

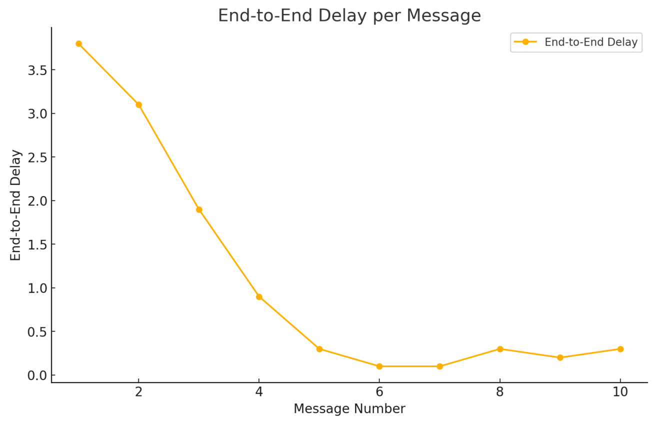

Fig. 3. DSR End-to-end delay.

Figure 3 shows the end-to-end delay for the ten messages. As can be seen, the nodes gradually show less delay. This happens because RREP messages from the first RREQ messages will cache the route. Eventually, the DSR will use a cached route and discard random broadcasts.

Figure 4 shows the routing overhead for the ten messages. As can be seen, the nodes gradually have less overhead. This happens because RREP messages from the first RREQ messages will cache the route. Eventually, the DSR will use a cached route and discard random broadcasts.

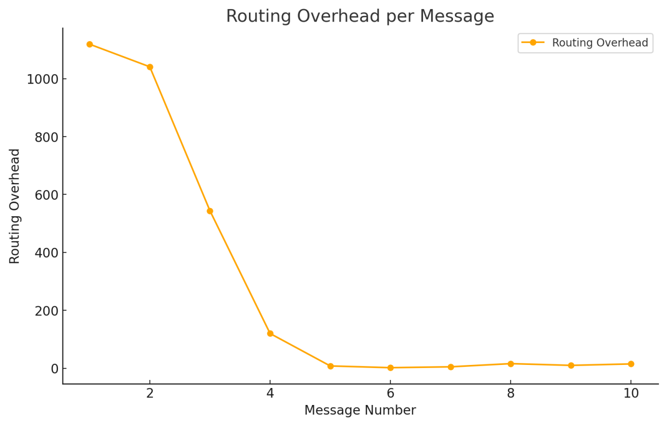

Fig. 4. DSR Routing Overhead.

IV. Translating DSR to SPIN model

Translating the Dynamic Source Routing (DSR) Network protocol from OMNeT++ to the SPIN model checker requires the behavior representation of the nodes, topology, and connections in the Promela language.

There is a step-by-step translation with code snippets:

A. SPIN components

In SPIN, DSR components are represented as follows:

- **Nodes:** Represented using `proctype`.
- **Messages:** Represented using `chan`.
- **Routing Logic:** Implemented using sending RREQ and RREP messages to neighbour node channel.

Translated OMNeT++ parameters and constants are translated to Promela:

```
#define GRID_SIDE 20
#define NODES (GRID_SIDE * GRID_SIDE)
#define MAX_ROUTE_LENGTH 10
#define MESSAGE_COUNT 10
```

The translated OMNeT++ message types and communication gates, where mtype, int, int, route_t stands for message type, source, message id and message route data structure respectively:

```
mtype = { RREQ, RREP };
typedef route_t {
    int path[MAX_ROUTE_LENGTH];
    int length;
}
chan channels[NODES] = [10] of { mtype, int,
    int, route_t };
```

Now, we can move to running of the simulation in Promela. For that, we will run each process in the init and initialize the source with destination nodes:

```
init {
    atomic {
        byte i = 0;
        network[0]!RREQ, 0, NODES - 1, 0; //
            setting source and destination
            nodes
        do
        :: i < NODES ->
            run Node(i);
            i++;
        :: else -> break;
        od;
    }
}
```

There is sendRREQ implementation in the promela of the DSR protocol. As Promela does not support multi-dimensional arrays, the nodes were flatted in an array of GRID_SIZE * GRID_SIZE size. That is why, there an algorithm to retrieve available neighbors' indexes and send the RREQ message through channels:

```
inline sendRREQ(sender, msgID, route) {
    int neighbors[4];
    neighbors[0] = -1;
    neighbors[1] = -1;
    neighbors[2] = -1;
    neighbors[3] = -1;

    int count = 0;

    // Left neighbor
    if
    :: (sender % GRID_SIDE != 0) ->
        neighbors[count] = sender - 1;
        count++
    :: else -> skip
    fi;

    // Right neighbor
    if
    :: ((sender + 1) % GRID_SIDE != 0) ->
        neighbors[count] = sender + 1;
        count++
    :: else -> skip
    fi;

    // Top neighbor
    if
    :: (sender >= GRID_SIDE) ->
        neighbors[count] = sender - GRID_SIDE;
        count++
    :: else -> skip
    fi;

    // Bottom neighbor
    if
    :: (sender < (GRID_SIDE * (GRID_SIDE - 1))
        ) ->
        neighbors[count] = sender + GRID_SIDE;
        count++
    :: else -> skip
    fi;

    int neighborIndex = 0;
    do
    :: neighborIndex < 4 ->
        int neighbor = neighbors[neighborIndex
            ];
        if
        :: neighbor != -1 ->
            channels[neighbor]!RREQ(sender,
                msgID, route);
        :: else -> skip
        fi;
        neighborIndex++
    :: else -> break
    od;
}
```

In the handleRREQ implementation, receivedRREQs was used to prevent duplicated messages similarly to in OMNeT++ simulation. Once the message was received, its route was updated. And if the RREQ message in the destination node RREP message is sent. Additionally, for the further LTL statement which will verify that RREQ reached the destination message index was saved in rrepSent.

```
inline handleRREQ(sender, receiver, msgID,
    route, receivedRREQs) {
    if
    :: msgID >= 0 && msgID < MESSAGE_COUNT &&
        receivedRREQs[msgID] ->
        skip;
    :: else ->
        receivedRREQs[msgID] = true;

        route_t routeCopy;
        i = 0;

        do
        :: i < route.length ->
            routeCopy.path[i] = route.path[i];
            i++
        :: else -> break
        od;

        routeCopy.length = route.length;
        routeCopy.path[routeCopy.length] =
            receiver;
        routeCopy.length++;

        if
        :: receiver == NODES - 1 ->
            rrepSent[msgID] = true;
            sendRREP(receiver, msgID,
                routeCopy);
        :: else -> sendRREQ(receiver, msgID,
            routeCopy);
        fi;
    fi;
}
```

Regarding sendRREP and handleRREP, it is sending RREP messages back to the source using the route and caching the route to the destination for future messages.

```
inline handleRREP(sender, receiver, msgID,
    route) {
    // Cache the full route
    int routeIndex = 0;
    do
    :: routeIndex < route.length ->
        routeCache[receiver].path[routeIndex]
            = route.path[routeIndex];
        routeIndex++
    :: else -> break
    od;

    routeCache[receiver].length = route.length
        ;
    sendRREP(receiver, msgID, route);
}
```

Finally, each node was implemented using proctype to cache the messages and forward them using handleRREQ and handleRREP.

```
proctype Node(int id) {
    bool isSource = (id == 0);
    bool isReceiver = (id == NODES - 1);
    int messageCounter = 0;
    bool receivedRREQs[MESSAGE_COUNT];


    int sender;
    int receiver;
    int msgID;
    route_t route;

    int i = 0;
    do
    :: i < MESSAGE_COUNT ->
        receivedRREQs[i] = false;
        i++
    :: else -> break
    od;

    do
    :: isSource && messageCounter <
       MESSAGE_COUNT ->
        route_t newRoute;
        newRoute.length = 1;
        newRoute.path[0] = id;
        sendRREQ(id, messageCounter, newRoute)
            ;
        messageCounter++;
    :: channels[id] ? RREQ(sender, msgID,
       route) ->
        handleRREQ(sender, id, msgID, route,
            receivedRREQs);
    :: channels[id] ? RREP(sender, msgID,
       route) ->
        handleRREP(sender, id, msgID, route);
    od;
}

init {
    atomic {
        int i = 0;
        do
        :: i < NODES ->
            run Node(i);
            i++
        :: else -> break
        od;
    }
}
```

B. *Verification using LTL statements*

The verification of the DSR network protocol was implemented using LTL statements. The LTL formulas ensure key properties of the OMNeT++ simulation such as RREP message delivery back to the source, deadlock freedom, and message acknowledgment that RREQ messages reached their destination and generated RREP message back to the source.

- **Message Delivery:** This LTL statement ensures that eventually we will get RREP message in the source node:

```
        ltl message_delivery { [](channels
            [0]?[RREP]) }
```

- **Deadlock Freedom:** This LTL statement ensures that the model will never stuck in progress:

```
    ltl deadlock_free { [](true) }
```

- **Message Reaches Destination:** This LTL statement ensures that every RREQ message eventually reaches the destination node, and upon reaching the destination, an RREP is generated and sent back:

```
    ltl message_reached_destination_ { []
        (rrepSent[0] || rrepSent[1] ||
        rrepSent[2] || rrepSent[3] ||
        rrepSent[4] || rrepSent[5] ||
        rrepSent[6] || rrepSent[7] ||
        rrepSent[8] || rrepSent[9]) }
```

These LTL statements are verified using SPIN model checker. They ensure the correctness of the DSR protocol implementation helping to adhere to the system's intended design.

V. Challenges

The Promela language is limited to its libraries and supported data structures. That is why the nodes array was flattened instead of using a multidimensional array, which simplifies the development of the algorithms. Additionally, there are no automatic tools that can translate the system code to the Promela model checker. That is why, it is hard to implement exact initial code and the current implementation of the DSR protocol in Promela language is simplified to catch only important behavior.

VI. Summary

In summary, the implementation of the DSR network in OMNeT++ was verified by using the SPIN model checker. The behavior was successfully translated into the Promela language to capture the system's DSR protocol simulation. Finally, the LTL statements provide measurement of the key properties such as RREQ message delivery to the destination, and RREP message delivery back to the source from the destination and guarantee that the system stays alive and no deadlock occurs. In other words, it was shown that the SPIN model checker can successfully verify network protocol implementations.