Mojgan Kamali | Luigia Petre

Modelling Link State Routing in Event-B

Turku Centre *for* Computer Science

TUCS Technical Report
No 1154, January 2016

Modelling Link State Routing in Event-B

Mojgan Kamali
Åbo Akademi University, Faculty of Science and Engineering,
the Agora building, 3rd floor, room 341A, Vesilinnantie 5, 20500 Turku, Finland
mojgan.kamali@abo.fi
Luigia Petre
Åbo Akademi University, Faculty of Science and Engineering,
the Agora building, 3rd floor, room 340I, Vesilinnantie 5, 20500 Turku, Finland
luigia.petre@abo.fi

TUCS Technical Report
No 1154, January 2016

Abstract

In this paper we present a stepwise formal development of the Optimised Link State Routing (OLSR) protocol in Event-B. OLSR is a proactive routing protocol which finds routes for different destinations in advance. As a consequence, whenever a data packet is injected into the network, aiming for a certain destination, it can be delivered immediately. To achieve this, routing tables in OLSR are continuously kept up-to-date, by following a rather complicated algorithm. By modelling OLSR in Event-B, we structure the OLSR complexity in five distinct abstraction layers that are manageable to understand and verify and are linked to each other by refinement. As Event-B is supported by a theorem proving platform (Rodin), we model and prove properties about OLSR in an automated and interactive manner, at a highly general level. Our approach can serve as a proof-of-concept to be adapted to other routing protocols for large-scale networks.

Keywords: Formal Analysis, Refinement, Event-B, link state routing

TUCS Laboratory
Distributed Systems Laboratory

1 Introduction

Wireless technologies are on the rise, ranging from laptops and smart phones that make work and connections easier, to sensor networks that produce and manipulate large amounts of data. In this study, we focus on contemporary wireless networks, in particular Wireless Mesh Networks (WMNs). WMNs are self-healing and self-organising wireless technologies supporting broadband communication without requiring any wired infrastructure. These networks are employed in a wide range of application areas such as emergency response networks, communication systems, video surveillance, etc. Due to these critical applications, correct behaviour and functioning of such networks should be guaranteed. One of the key factors determining the performance and reliability of WMNs is their routing protocols.

The fundamental role of (WMN) routing protocols is to provide routes between nodes for communication. These protocols disseminate information through the network in order to select such routes and thus enable nodes to send data packets to arbitrary destinations in the network. Routing protocols for WMNs are either proactive or reactive. Proactive protocols attempt to select routes in advance, by exchanging control messages about all the other nodes of the network. Consequently, an injected data packet can be delivered to the destination immediately. Examples of such protocols are Optimised Link State Routing (OLSR) protocol, Better Approach To Mobile Ad hoc Networking (BATMAN) routing protocols, etc. Reactive protocols search for routes to destination nodes on demand, whenever a data packet is injected into the network. Examples of reactive protocols are Ad hoc On-Demand Distance Vector (AODV) protocol, Dynamic Source Routing (DSR) protocol, etc.

In this paper we focus on the OLSR protocol [9]. We formally model this protocol through incremental stepwise refinements in Event-B [2], a formal modelling approach which provides automatic tool support for modelling and proving various properties.

OLSR is a rather complicated protocol. This is mainly due to its proactive nature, requiring it to keep up-to-date information about routes to all destinations from any node. Assumed to work in the WMN setup, this implies that all routes are checked periodically, to prevent the case of various links going down. Indeed, the OLSR specification [9] prescribes certain routines to be performed every 2 seconds, and some other every 5 seconds. There are two kinds of control messages exchanged by nodes for updating the complex structure of the routing tables. By stepwise modelling OLSR in Event-B we gain a deep understanding of the OLSR mechanisms at five different layers of abstraction. The first two layers simply model a routing protocol, first more abstractly and then more concretely, modelling only very abstractly the proactive OLSR behaviour. The following two layers add the infrastructure necessary for modelling the proactive behaviour, again in a more abstract manner first and somewhat more concretely

in layer four. In the final layer we add the defining characteristic of OLSR, of selecting only specific nodes to broadcast control messages, so as not to strangle the network traffic. These layers are related by refinement: this means that the properties and the behaviour of any model are kept in all its subsequent refinements. Hence, we can prove certain properties in a more abstract layer (when the property is simpler to prove) and then develop the models in the more concrete layers so that they do not break those proven properties. Our main contribution is to demonstrate the management of complexity in an elegant manner and to offer our solution as a proof-of-concept for other routing protocols. As a consequence of the specific properties we prove, we discover that OLSR does not find optimal routes to all the destinations, while also proving that it indeed discovers routes to all the destinations as well as it delivers data packets on these discovered routes.

We proceed as follows. In Section 2 we briefly overview the OLSR protocol and in Section 3 we outline our modelling framework Event-B. In Section 4 we present our formal development of the OLSR protocol. Verification results and the impact of our paper are described in Section 5. We discuss related work in Section 6 and draw some conclusions in Section 7.

2 Overview of the OLSR protocol

Optimised Link State Routing (OLSR) is a proactive routing protocol adapted for WMN applications. (Data) packets need to be transmitted from certain sources to their destinations in a WMN, and as a routing protocol, OLSR has to find routes for these packets. OLSR has all the routes available (in its constantly updating routing tables of all nodes) when required. This is achieved by exchanging control messages among nodes. The OLSR protocol is an optimisation over other proactive routing protocols, as it decreases the traffic in the network by selecting some nodes as the so-called MultiPoint Relays (MPRs); only these nodes are then able to transmit control messages. Every node a has a set of one-hop neighbour nodes as MPRs, connected to the two-hop neighbours of node a.

This protocol does not depend on any central entity for coordination and is working in a completely distributed manner. The OLSR protocol broadcasts two types of control messages, HELLO messages and TC (Topology Control) messages, to distribute information about the nodes of the network. A HELLO message is broadcast every 2 seconds on single hops (to one-hop neighbours and not forwarded). It is used to detect one-hop and two-hop neighbours of every node and, based on this, to select MPR nodes. A TC message is broadcast every 5 seconds to build and update topological information. The TC messages are broadcast and forwarded via MPRs only.

Each node has a routing table to store and keep the information about all the other destinations in the network. When a node receives HELLO or TC messages, it updates its routing table based on the information in the control messages. As a

consequence, the topological information about different destination nodes is continuously updated in the routing tables, making it feasible to deliver data packets to arbitrary destination nodes.

3 Overview of Event-B

Event-B [2] is a formal method based on the B-Method [1] and on the Action Systems [5] framework for modelling and analysing distributed, parallel and reactive systems. Automated support for modelling and verification is provided by the Rodin Platform [3]. Event-B uses *context* and *machine* modules to define system specifications (models) and to express behaviour and properties of systems. An Event-B context contains constants and carrier sets, whose properties are expressed as a list of axioms for the model. Thus, a context contains the static part of the system. A machine contains the dynamic part of the system. The relationship between a machine and its context is defined by the keyword *Sees* showing that the machine can have access to the contents of the context.

A machine describes the state of the model with *variables* that are updated by *events*. Events may contain *guards*: these are logical properties linking constants, variables and potentially local variables of the event. The guards must evaluate to *true* for the event to be *enabled*. If several events are enabled simultaneously, only one event is chosen for execution, non-deterministically. An Event-B machine also contains *invariants*. These are expressed in terms of constants and variables of the context/machine. Invariants are properties that need to hold for any reachable state of the model. This means that invariants must hold before and after any occurrence of any event.

The main development strategy in Event-B is that of *refinement*. A machine A can be refined by a machine B (denoted $A \sqsubseteq B$) when A's behaviour is not altered by B in any way and new variables are added in B, together with new events for updating these variables. Events in machine A can also be refined in machine B, by enriching their behaviour with behaviour of the new variables, for instance. This type of refinement is called *superposition* refinement. Other types of refinement also exist, for instance *data refinement*, where some variables in the more abstract machine are replaced by other variables in the refined machine; in this case, a so-called *gluing invariant* is specified, that describes the relation between the old and the new variables. All the occurrences of the old variables are replaced in events in the refined machine by the new variables; the only place where the old variables still appear is in the gluing invariant. To prove that machine A is refined by machine B, a set of so-called *proof obligations* is generated by the Rodin platform. Some are discharged automatically by Rodin and some need to be discharged interactively with help from the modeller.

All events in the abstract model are kept through in the refinement chain. Some events are refined, and then the refined event replaces its more abstract counter-

part. Even if we do not show the abstract events from previous levels in the paper (due to lack of space) they are still part of the refined model.

4 Event-B Model of OLSR

In this section, we express the OLSR protocol development at five different layers of abstraction, overviewed below. We summarise our development of the OLSR model in Fig. 1, where we illustrate the five abstraction layers containing the corresponding five machines as well as two contexts.

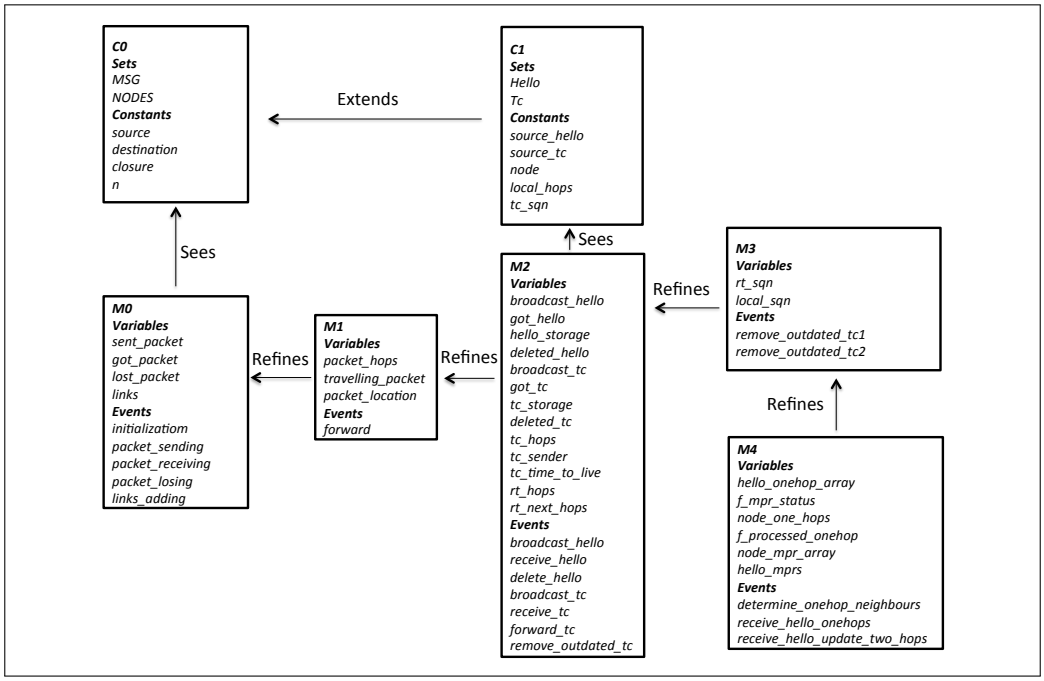

Figure 1: Overview of model development

Initial Model (*M0*): Basic routing protocol behaviour, i.e., sending, receiving, and losing data packets, is specified, together with an abstraction of proactive routing behaviour.
Refinement 1 (*M1*): A storing and forwarding architecture for data packets from a source node to a destination node is modelled in this step.
Refinement 2 (*M2*): The basic behaviour of route discovery is introduced in this refinement. This level describes the essential OLSR behaviour for sending and receiving control messages based on which each node updates its routing table.
Refinement 3 (*M3*): More detailed information about the route discovery protocol and how to process new control messages are provided in this level of refinement.
Refinement 4 (*M4*): Selection of MPR nodes and how they help to decrease the traffic in the network are introduced in this step.

The refinement relation between machines is emphasised in Fig. 1 and we illustrate the *new* variables and events in all the machines, together with constants and carrier sets in the contexts. In the following sections we describe these entities in detail. The reader can also consult our models in Event-B or in pdf format at `http://users.abo.fi/mokamali/ICFEM_2016`.

4.1 The Context *C0* and Initial Model *M0*

In context *C0* we define two carrier sets *NODES* and *MSG* and four constants *source*, *destination*, *closure*, and *n*. The carrier set *NODES* models the nodes in the network; this set is finite and non-empty (modelled by *axm1* and *axm2*, respectively). The number of elements in *NODES* is equal to *n* (*axm3*) which is larger than 1 (*axm4*). In axioms *axm5* and *axm6*, we define the carrier set *MSG* as a finite and non-empty set, modelling the set of all (user) data packets. In axioms *axm7* and *axm8*, we define the type of the constants *source* and *destination*, modeling the source and, respectively, the destination of all data packets; these are total functions mapping *MSG* to *NODES*. The constant *closure* models the transitive closure of binary relations between nodes (*NODES*) in axioms *axm9–axm12*. The sets, constants and axioms belong to the context *C0*, seen by our initial model *M0*.

AXIOMS

MSG
$NODES$

CONSTANTS

$source$
$destination$
$closure$
n

AXIOMS

`axm1` : $finite(NODES)$
`axm2` : $NODES \neq \varnothing$
`axm3` : $card(NODES) = n$
`axm4` : $n > 1$
`axm5` : $finite(MSG)$
`axm6` : $MSG \neq \varnothing$
`axm7` : $source \in MSG \rightarrow NODES$
`axm8` : $destination \in MSG \rightarrow NODES$
`axm9` : $closure \in (NODES \leftrightarrow NODES) \rightarrow (NODES \leftrightarrow NODES)$
`axm10` : $\forall r \cdot r \subseteq closure(r)$
`axm11` : $\forall r \cdot closure(r); r \subseteq closure(r)$
`axm12` : $\forall r, s \cdot r \subseteq s \wedge s; r \subseteq s \Rightarrow closure(r) \subseteq s$

In *M0*, we model an abstract version of the OLSR protocol. We have four variables, namely *sent_packet*, *lost_packet*, *got_packet* and *links*. Variable *sent_packet* is a subset of *MSG* (*inv1*) modelling the packets actually sent through the network

(injected). Lost packets and received packets in the network are modelled by variables *lost_packet* and *got_packet*, respectively. The set of lost packets (*lost_packet*) and received packets (*got_packet*) are subsets of the set of all injected packets (*sent_packet*); this is a safety property modelled by invariants *inv2* and *inv3*. A data packet cannot be received and lost at the same time as modelled in *inv4*. Variable *links* models the current links in the network (*inv5*). No node is connected to itself, as modelled by invariant *inv6*.

INVARIANTS

$inv1$: $sent\_packet \subseteq MSG$
$inv2$: $lost\_packet \subseteq sent\_packet$
$inv3$: $got\_packet \subseteq sent\_packet$
$inv4$: $got\_packet \cap lost\_packet = \varnothing$
$inv5$: $links \in NODES \leftrightarrow NODES$
$inv6$: $(NODES \lhd id) \cap links = \varnothing$

There are four simple events in our abstract model in addition to the initialisation event where all variables get value $\varnothing$. The event *packet_sending* models the sending of a data packet *msg* not yet injected from a source node *s* to a destination node *d* (*grd1–grd3*). The main guard of this event ensures that there is a path from *s* to *d* (*grd4*). If theses conditions hold, then *msg* can be sent (injected in the network to eventually make its way to *d*).

Event *packet_sending* $\widehat{=}$
any
$s,\ d,\ msg$
where
$grd1$: $msg \in MSG \land msg \notin sent\_packet$
$grd2$: $source(msg) = s \land destination(msg) = d$
$grd3$: $s \neq d$
$grd4$: $s \mapsto d \in closure(links)$
then
$act1$: $sent\_packet := sent\_packet \cup \{msg\}$
end

Event *packet_receiving* models the successful receiving of the data packet *msg* by a destination node. The guard of this event (*grd1*) models that *msg* has not been received or lost yet.

Event *packet_receiving* $\widehat{=}$
any
msg
where
$grd1$: $msg \in sent\_packet \setminus (got\_packet \cup lost\_packet)$
then
$act1$: $got\_packet := got\_packet \cup \{msg\}$
end

Event *packet_losing* models loss of data packets. The guard of this event (*grd1*) models that *msg* has not been received or lost yet.

Event *packet_losing* $\widehat{=}$
any
msg
where
grd1 : $msg \in sent\_packet \setminus (got\_packet \cup lost\_packet)$
then
act1 : $lost\_packet := lost\_packet \cup \{msg\}$
end

Event *links_adding* models that some arbitrary links not yet in the network (*grd1*) may come up; these new links are added to the set *links*.

Event *links_adding* $\widehat{=}$
any
$s,\ d$
where
grd1 : $s \mapsto d \notin links$
grd2 : $s \neq d$
then
act1 : $links := links \cup \{s \mapsto d\}$
end

We note here that the first three events of *M0*, *packet_sending*, *packet_receiving* and *packet_losing* are events common to any routing protocol. The specific proactive feature that routes to destinations are continuously updated in the routing table, i.e., the valid links are continuously updated, is modelled in *M0* abstractly, with our fourth event *link_adding*. Later in the refinement chain, adding links in the network will be replaced by updating routing tables based on the information received from HELLO and TC messages. At this abstract level however, we only have data packets as messages. Control messages are introduced in *M2*. We also note that receiving data packets is magical, with having no intermediate nodes in between the source and the destination of a data packet. A sent data packet can be either received or lost in a non-deterministic manner. We add intermediate nodes in between sources and destinations in *M1*.

4.2 First Refinement *M1*: Storing and Forwarding Architecture

In the initial model, data packets are received in an atomic magical step from a source node to a destination node. This is of course not the case in a real protocol. Data packets are transferred using multi-hop communication and they are forwarded hop by hop from a source node *s* to destination node *d*. Hence, in this refinement step, we model the storing and forwarding architecture of data packets when not all the nodes are directly connected and the data packet has to be forwarded by several intermediate nodes before being delivered at the destination node. For this, we define three new variables and one new event. The new variable *travelling_packet* is a subset of *sent_packet* (*inv1*), modelling the packets that are sent but not yet received or lost. If we study *M0*, we note that there are data

packets in *sent_packet* that are not lost nor received: we refer to them in the guards of events *packet_receiving* and *packet_losing*. In *M1* this set of messages is made concrete by the new variable *travelling_packet*.

Invariant *inv2* states that a data packet can either be travelling, received or lost (sets *travelling_packet*, *got_packet* and *lost_packet* do not have any elements in common). Invariant *inv3* models that the *sent_packet* set is partitioned into the subsets *travelling_packet*, *got_packet* and *lost_packet*. Related to *travelling_packet*, variable *packet_location* is a binary relation between *sent_packet* and *NODES* (*inv4*). This variable models the node in the network where a data packet is currently located at. If a data packet is not sent, then it cannot be received by or located in any nodes of the network (*inv5*). If a data packet is received, the location of the packet must be the destination of that packet (*inv6*). A node cannot have contradictory information about a data packet, i.e., a data packet cannot be located at two different nodes (*inv7*).

INVARIANTS

inv1 : $travelling\_packet \subseteq sent\_packet$

inv2 : $travelling\_packet \cap got\_packet \cap lost\_packet = \varnothing$

inv3 : $travelling\_packet \cup got\_packet \cup lost\_packet = sent\_packet$

inv4 : $packet\_location \in sent\_packet \leftrightarrow NODES$

inv5 : $\forall m \cdot m \in MSG \wedge m \notin sent\_packet \Rightarrow (m \notin got\_packet \wedge (\forall s \cdot s \in NODES \Rightarrow m \mapsto s \notin packet\_location))$

inv6 : $\forall m \cdot m \in got\_packet \Rightarrow m \mapsto destination(m) \in packet\_location$

inv7 : $\forall m, s, r \cdot m \mapsto s \in packet\_location \wedge m \mapsto r \in packet\_location \Rightarrow s = r$

inv8 : $packet\_hops \in MSG \rightarrow \mathbb{N}$

inv9 : $\forall m \cdot m \in sent\_packet \setminus (lost\_packet \cup got\_packet) \Leftrightarrow m \in travelling\_packet$

inv10 : $dom(packet\_location) = sent\_packet$

inv11 : $sent\_packet \setminus lost\_packet = got\_packet \Leftrightarrow travelling\_packet = \varnothing$

The third new variable in *M1*, *packet_hops*, is described by *inv8*: it models the number of hops a data packet is forwarded and is another element of proactive routing added here as a natural companion to the *packet_location* variable. Invariant *inv9* models that a sent packet that is not lost and not received must be travelling (and vice versa). This is a reformulation of invariants *inv2* and *inv3*, useful for proving various obligations. Invariant *inv10* guarantees that all the packets located at different nodes have been sent already. Invariant *inv11* describes the essential liveness property of the routing protocol: when all the sent (and not lost packets) are received, then there is no travelling packet in the network and vice versa. This means that eventually all non-lost packets reach their destinations.

The new event *forward* of *M1* models the transfer of the data packets between two connected nodes. The first guard states that a data packet (*msg*) is travelling through the network. The second guard models that the destination node of data packet *msg* is *d*. The last three guards, i.e., *grd3–grd5*, introduce the intermediate node *a*, that is not the destination node, but the data packet is stored at *a*, and not in the intermediate node *b*. There is a route from the current intermediate node *a*

to destination d (*grd4*) and node b is a neighbour of a (*grd5*). Then, the number of hops of the packet is increased by 1 and the data packet is forwarded to node b. Notation $\lhd\!\!+$ denotes the relation overriding ($r \lhd\!\!+ t = t \cup (dom(t) \lhd r)$).

Event *forward* $\widehat{=}$

any

$d,\ a,\ b,\ msg$

where

`grd1` : $msg \in travelling\_packet$
`grd2` : $destination(msg) = d$
`grd3` : $a \neq destination(msg) \wedge msg \mapsto a \in packet\_location$
$\wedge\ msg \mapsto b \notin packet\_location$
`grd4` : $a \mapsto d \in closure(links)$
`grd5` : $a \mapsto b \in links$

then

`act1` : $packet\_location := (packet\_location \setminus \{msg \mapsto a\}) \cup \{msg \mapsto b\}$
`act2` : $packet\_hops := packet\_hops \lhd\!\!+ \{msg \mapsto packet\_hops(msg) + 1\}$

end

There new actions *act5*, *act6* and *act7* are added in the initialisation event to assign values $MSG \times \{0\}$, $\varnothing$ and $\varnothing$ respectively to the new variables *packet_hops*, *travelling_packet* and *packet_location*. Three new actions *act2*, *act3* and *act4* are added in *packet_sending* event, to add $msg \mapsto 0$ to *packet_hops*, *msg* to *travelling_packet* and $msg \mapsto source(msg)$ to *packet_location*.

Guard *grd1* in *M0* is modified in *M1* with respect to *packet_receiving* event to $msg \in travelling\_packet$: this expresses exactly that *msg* is sent but not yet received nor lost (as deducible from *inv9* or invariants *inv2* and *inv3*). New guards *grd2* and *grd3* are added: guard *grd2* models that the destination of the message (*msg*) is d and *grd3*: $msg \mapsto d \in packet\_location$, models that only a data packet located already at its destination can be received. Corresponding action *act2* is added in this event to remove *msg* from *travelling_packet*. Thus, the receiving is not magical anymore, but only when the data packet has arrived at its destination. At that moment, the data packet changes its state from travelling (i.e., sent but not yet received nor lost) to received.

Guard *grd1* in *M0* is modified in *M1* with respect to event *packet_losing*, exactly similar to the modification in *packet_receiving* explained above. Four new guards *grd2–grd5* are added in *packet_losing* to model the location s where the data packet currently is. While in *M0* the packet losing was non-deterministic, now it is for a certain reason captured in these new guards:

- $destination\ (msg) = d$
- $msg \mapsto s \in packet\_location$
- $s \neq d$
- $s \mapsto d \notin closure\ (links)$

Hence, in *M1* we lose a message where there is no route from its current location s to its destination d. The extra *act2* is added in the event to remove *msg* from *travelling_packet* since the data packet *msg* will not be travelling anymore.

We also note that when forwarding a message (event *forward*) we do not check that $a \mapsto b$ belongs to a proper route from a to d; b only has to be a neighbour of a but it can well be in the wrong direction.

4.3 Second Refinement *M2*: Route Discovery Protocol

The route discovery protocol is the most important and complicated refinement step of this model. In this level of refinement, we investigate whether or not nodes can find optimal routes to different destination nodes. We add OLSR control messages (HELLO and TC) and model the routing tables of every node. In this step, we replace the centralised functioning of the routing protocol in models *M0* and *M1* with a distributed functioning by data refinement. We explain all these steps in detail in the following.

We extend the context *C0* to context *C1* that adds two carrier sets *Hello* and *Tc* for modelling the control messages of OLSR, i.e., HELLO and TC messages, respectively. Two additional constants *source_hello* and *source_tc* are modelled as total functions mapping respectively sets *Hello* and *Tc* to the set *NODES*. These constants model the source nodes of all these control messages (*axm1* and *axm2*). The constant *node* denotes an imaginary node, not in *NODES* (*axm3*). It is used to denote initially the next node to take for any route. Constant *tc_sqn* models the sequence number of each TC message used for denoting the freshness of the message; *tc_sqn* must be a natural number larger than 0 (*axm4* and *axm5*). Constant *tc_sqn* is only used in *M3*.

AXIOMS

axm1 : $source\_hello \in Hello \rightarrow NODES$
axm2 : $source\_tc \in Tc \rightarrow NODES$
axm3 : $node \notin NODES$
axm4 : $tc\_sqn \in Tc \rightarrow \mathbb{N}$
axm5 : $\forall tc \cdot tc \in Tc \Rightarrow tc\_sqn(tc) > 0$

Thirteen new variables are defined to model the behaviour of the OLSR protocol. As mentioned in Section 2, nodes broadcast HELLO messages in the network. The broadcast HELLO messages are denoted by variable *broadcast_hello* and are a subset of all *HELLO* messages (*inv1*). Some broadcast messages are deleted, denoted by *deleted_hello* and some broadcast messages are received, denoted by *got_hello* (*inv2* and *inv3*). HELLO messages can be located at various nodes in the network; the relation between a HELLO message and its current location is modelled by variable *hello_storage* (*inv4*). Similar sets of variables are introduced for TC messages, namely *broadcast_tc*, *got_tc*, *deleted_tc* and *tc_storage*, respectively (*inv5, inv6, inv7, inv8*). Invariants *inv16 – inv19* model that *broadcast_hello* and

broadcast_tc are partitioned into sets *got_hello*, *deleted_hello*, *ran(hello_storage)* and *got_tc*, *deleted_tc*, *ran(tc_storage)*, respectively. Invariant *inv20* models that if a TC message has been broadcast and it is not located at some intermediary nodes and it does not belong to the *deleted_tc* set, then it is received.

Information in the TC messages is modelled by variables *tc_hops*, *tc_sender* and *tc_time_to_live* showing how many times a TC message has been forwarded, the sender of the TC message, and how many hops a TC message can be forwarded, respectively. Invariants *inv9 – inv11* model the types of these variables. Invariants *inv12* and *inv13* model the routing table of each node in the network containing the distance to other nodes of the network (*rt_hops*) and the next nodes along the paths to different destination nodes (*rt_next_hops*). As we have reasons to believe that OLSR is a best-effort protocol, we have added invariant *inv14* to our *M2* model. This essentially describes the existence of non-optimal deliveries of data packets: the actual number of hops that *msg* passed through is bigger than what the routing tables would have recommended. In other words, data packet *msg* arrived on a longer path to its destination. This happens due to the distributed nature of OLSR: in event *forward* we forward a data packet to any neighbour, not checking if this neighbour is in the suitable direction toward destination. To be able to check that, one would need a global view about the network, i.e., a centralised approach.

If a TC message from one originator is received by a node and the hops of the TC message is 0, then the originator and the sender of the message are the same and the originator and the receiving node are one-hop neighbours (*inv15*).

The functioning of the routing protocol in *M0* and *M1* is centralised because we check in various events' guards, conditions about paths among any two nodes in the network (using the *closure* property). Only a centralised algorithm, that has access to all the network nodes, can calculate the closure of any link. OLSR is, however, a distributed algorithm based on nodes exchanging (HELLO and TC) messages among each other, in order to find out about active links. We have now modelled these messages, so all the guards referring to *links* and *closure(links)* in *M0* and *M1* are replaced in *M2* with guard expressed in terms of the routing table. This operation is a data refinement, and to be able to perform it, we need some gluing invariants relating variable *links* in *M1* to variable *rt_next_hops* in *M2*. The variable *links* thus appears in *M2* only in the gluing invariants *inv21* and *inv22*. If two nodes a and b are connected ($a \mapsto b \in links$), this means that the next node in the routing table of b for node a is a itself (*inv21*). Invariant *inv22* states that if there is a path from one node to the destination node, then the routing table entry corresponding to the next node along the path to the destination node of that node must have been updated (*rt_next_hops(s)(destination (m)* $\neq$ *node*). Invariant *inv22* thus models route discovery, essentially expressing that, if there is a path from source to destination, then the protocol discovers it.

The last invariant *inv23* deals with the data packet loss, meaning that if a data packet is located at a node and that node has not updated the information about

the destination node of the data packet, then the data packet is not received and it will be lost.

INVARIANTS

inv1 : $broadcast\_hello \subseteq Hello$
inv2 : $got\_hello \subseteq broadcast\_hello$
inv3 : $deleted\_hello \subseteq broadcast\_hello$
inv4 : $hello\_storage \in NODES \leftrightarrow Hello$
inv5 : $broadcast\_tc \subseteq Tc$
inv6 : $got\_tc \subseteq broadcast\_tc$
inv7 : $deleted\_tc \subseteq broadcast\_tc$
inv8 : $tc\_storage \in NODES \leftrightarrow Tc$
inv9 : $tc\_hops \in Tc \rightarrow \mathbb{N}$
inv10 : $tc\_sender \in Tc \rightarrow NODES$
inv11 : $tc\_time\_to\_live \in Tc \rightarrow \mathbb{N}$
inv12 : $rt\_hops \in NODES \rightarrow (NODES \rightarrow \mathbb{N})$
inv13 : $rt\_next\_hops \in NODES \rightarrow (NODES \rightarrow NODES)$
inv14 : $\exists msg, s \cdot s = source(msg) \land msg \in got\_packet \Rightarrow packet\_hops(msg) > rt\_hops(s)(destination(msg))$
inv15 : $\forall a, tc \cdot a \mapsto tc \in tc\_storage \land tc\_hops(tc) = 0 \Rightarrow source\_tc(tc) = tc\_sender(tc) \land rt\_hops(a)(source\_tc(tc)) = 1$
inv16 : $got\_hello \cup ran(hello\_storage) \cup deleted\_hello = broadcast\_hello$
inv17 : $got\_hello \cap ran(hello\_storage) \cap deleted\_hello = \varnothing$
inv18 : $got\_tc \cup ran(tc\_storage) \cup deleted\_tc = broadcast\_tc$
inv19 : $got\_tc \cap ran(tc\_storage) \cap deleted\_tc = \varnothing$
inv20 : $\forall tc \cdot tc \in broadcast\_tc \land tc \notin ran(tc\_storage) \land tc \notin deleted\_tc \Rightarrow tc \in got\_tc$
inv21 : $\forall a, b \cdot a \mapsto b \in links \Leftrightarrow rt\_next\_hops(b)(a) = a$
inv22 : $\forall s, m \cdot s \mapsto destination(m) \in closure(links) \Leftrightarrow rt\_next\_hops(s)(destination(m)) \neq node$
inv23 : $\forall s, m \cdot m \mapsto s \in packet\_location \land rt\_next\_hops(s)(destination(m)) = node \Leftrightarrow m \notin got\_packet \land m \in lost\_packet$

We define seven additional events to model the basic behaviour of OLSR. The first event *broadcast_hello* is used to model the broadcasting of HELLO messages in the network. This event is similar to the *packet_sending* event in *M1*.

Event *broadcast_hello* $\widehat{=}$
any
grd1 : $s \in NODES$
grd2 : $hello \in Hello$
grd3 : $hello \notin broadcast\_hello$
grd4 : $source\_hello(hello) = s$
grd5 : $s \mapsto hello \notin hello\_storage$
then
act1 : $broadcast\_hello := broadcast\_hello \cup \{hello\}$
act2 : $hello\_storage := hello\_storage \cup \{s \mapsto hello\}$
end

Event *receive_hello* refines event *links_adding* and models receiving of HELLO messages.While similar to the *packet_receiving* event in *M1*, here we also model the specific role of the HELLO messages in the OLSR protocol. Upon receiving a

HELLO message, the receiving node updates the corresponding routing table for the message originator and adds the HELLO message to the *got_hello* set.

Event *receive_hello* $\widehat{=}$
extends *links_adding*
any
$s,\ d,\ hello$
where
$grd1$: $hello \in broadcast\_hello \setminus (got\_hello \cup deleted\_hello)$
$\wedge\ source\_hello(hello) = s$
$grd2$: $s \neq d$
$grd3$: $d \mapsto hello \in hello\_storage$
then
$act1$: $rt\_hops(d) := rt\_hops(d) \lhd\!\!+ \{s \mapsto 1\}$
$act2$: $rt\_next\_hops(d) := rt\_next\_hops(d) \lhd\!\!+ \{s \mapsto source\_hello(hello)\}$
$act3$: $got\_hello := got\_hello \cup \{hello\}$
end

Event *delete_hello* removes HELLO messages from both sets *got_hello* and *hello_storage* and adds it into the set *deleted_hello*. Removing messages from *got_hello* and *hello_storage* is a feature of the OLSR protocol.

Event *delete_hello* $\widehat{=}$
any
$d,\ hello$
where
$grd1$: $d \in NODES$
$grd2$: $d \mapsto hello \in hello\_storage$
$grd3$: $hello \in got\_hello$
then
$act1$: $deleted\_hello := deleted\_hello \cup \{hello\}$
$act2$: $got\_hello := got\_hello \setminus \{hello\}$
$act3$: $hello\_storage := hello\_storage \setminus \{d \mapsto hello\}$
end

The next event *broadcast_tc* models the broadcast of TC messages through the network. This event adds the required information in the corresponding variable (*tc_sender*) and then broadcasts the message.

Event *broadcast_tc* $\widehat{=}$
any
$s,\ tc$
where
$grd1$: $s \in NODES$
$grd2$: $tc \in Tc$
$grd3$: $tc \notin broadcast\_tc$
$grd4$: $source\_tc(tc) = s$
$grd5$: $s \mapsto tc \notin tc\_storage$
then
$act1$: $tc\_sender := tc\_sender \lhd\!\!+ \{tc \mapsto s\}$
$act2$: $tc\_storage := tc\_storage \cup \{s \mapsto tc\}$
$act3$: $broadcast\_tc := broadcast\_tc \cup \{tc\}$
end

When a node receives a TC message from a node, it updates its routing table for the originator of the message and adds the messages into *got_tc* set. This is modelled by event *receive_tc* which also refines *links_adding*.

Event *receive_tc* $\widehat{=}$
extends *links_adding*
any
$s,\ d,\ tc$
where
grd1 : $tc \in broadcast\_tc \setminus (got\_tc \cup deleted\_tc)$
grd2 : $source\_tc(tc) = s \land source\_tc(tc) \neq d$
grd3 : $s \neq d$
grd4 : $d \mapsto tc \in tc\_storage$
then
act1 : $rt\_hops(d) := rt\_hops(d) \lhd\!\!+ \{s \mapsto tc\_hops(tc) + 1\}$
act2 : $rt\_next\_hops(d) := rt\_next\_hops(d) \lhd\!\!+ \{s \mapsto tc\_sender(tc)\}$
act3 : $got\_tc := got\_tc \cup \{tc\}$
end

As mentioned in Section 2, TC messages can be forwarded: event *forward_tc* models the forwarding of TC messages in the network. If the guards of this event evaluate to true, then the TC message can be forwarded. When a TC message is forwarded, its corresponding variables *tc_hops*, *tc_sender*, and *tc_time_to_live* are updated.

Event *forward_tc* $\widehat{=}$
any
$a,\ b,\ tc$
where
grd1 : $tc \in got\_tc \land source\_tc(tc) \neq a$
grd2 : $b \in NODES \land a \in NODES \land a \neq b$
grd3 : $a \mapsto tc \in tc\_storage \land b \mapsto tc \notin tc\_storage$
grd4 : $tc\_time\_to\_live(tc) > 1$
then
act1 : $tc\_storage := (tc\_storage \setminus \{a \mapsto tc\}) \cup \{b \mapsto tc\}$
act2 : $tc\_hops := tc\_hops \lhd\!\!+ \{tc \mapsto tc\_hops(tc) + 1\}$
act3 : $tc\_sender := tc\_sender \lhd\!\!+ \{tc \mapsto a\}$
act4 : $tc\_time\_to\_live := tc\_time\_to\_live \lhd\!\!+ \{tc \mapsto tc\_time\_to\_live(tc) - 1\}$
end

The last event (*remove_outdated_tc*) models the removal of old TC messages from the network. The guards of this event model that if a message has been received before and it is in the *tc_storage* of some node, then it can be deleted from the network. In *M3*, we refine this event to more precisely model the behaviour of OLSR w.r.t. when some TC messages must be removed from the network.

Event *remove_outdated_tc* $\widehat{=}$
any
$tc,\ a$
where
grd1 : $a \in NODES$
grd2 : $tc \in got\_tc$
grd3 : $a \mapsto tc \in tc\_storage$
then
act1 : $deleted\_tc := deleted\_tc \cup \{tc\}$
act2: $tc\_storage := tc\_storage \setminus \{a \mapsto tc\}$
act3 : $got\_tc := got\_tc \setminus \{tc\}$
end

In addition to all these new variables and events, the old events are slightly modified. Namely, we send and forward data packets (events *packet_sending* and *forward*, respectively) only if information in the corresponding routing table is recent, i.e., *rt_next_hops (s)(d)* $\neq$ *node*. This is because of the data refinement: we replace the guards expressed in terms of *link* and *closure(links)* by guards expressed in terms of the routing table variables. We also observe that event *links_adding* from *M0* and *M1* is now replaced by events *broadcast_hello* and *broadcast_tc*. Likewise, losing a data packet (event *packet_losing*) happens if the information in the routing table is not recent, i.e., *rt_next_hops (s)(d)* = *node*.

4.4 Third Refinement *M3*: Sequence Numbers

In this level of refinement, we define new variables to model sequence numbers and to avoid processing TC messages with old information. In machine *M2*, whenever a node receives a TC message, it just processes it without considering if the message has some new information or not. However in this model, we define sequence numbers to avoid this. When a node receives a TC message from an originator node, it stores the TC sequence number to the corresponding routing table so that whenever it receives another TC message from that originator, it compares the sequence number of the new TC message and the sequence number stored in the routing table; it thus figures out if the TC message must be processed or not. Invariant *inv1* is defined to model the routing table entries for sequence numbers (in variable *rt_sqn*). Invariant *inv2* models the local sequence number of each node in variable *local_sqn*. Whenever a TC message is broadcast, the node increases *local_sqn* by 1.

Invariant *inv3* checks if the sequence number of a TC message flooded in the network and located in a node is smaller than the sequence number of TC message originator in the routing table of the receiving node; in that case, the TC message must not be processed and it must be removed from the network.

INVARIANTS

inv1 : $rt\_sqn \in NODES \rightarrow (NODES \rightarrow \mathbb{N})$

inv2 : $local\_sqn \in NODES \rightarrow \mathbb{N}$

inv3 : $\forall tc, d \cdot tc \in broadcast\_tc \wedge d \mapsto tc \in tc\_storage \wedge$
$tc\_sqn(tc) \leq rt\_sqn(d)(source\_tc(tc)) \Rightarrow tc \notin got\_tc \wedge tc \in deleted\_tc$

We define a new action in event *braodcast_tc* which increases the value of *local_sqn* of each node after a TC message is broadcast. The sequence number of a TC message is checked upon receipt by a node (event *receive_tc*, *grd5*) to see if the TC message has been processed before or not. If not, then a new action *act4* is added to this event, that updates the routing table entry for sequence numbers w.r.t. the destination *d*.

Event *receive_tc* $\widehat{=}$
extends *receive_tc*
any
$s,\ d,\ tc$
where
$grd1$: $tc \in broadcast\_tc \setminus (got\_tc \cup deleted\_tc)$
$grd2$: $source\_tc(tc) = s \land source\_tc(tc) \neq d$
$grd3$: $s \neq d$
$grd4$: $d \mapsto tc \in tc\_storage$
$grd5$: $tc\_sqn(tc) > (rt\_sqn(d)(s))$
then
$act1$: $rt\_hops(d) := rt\_hops(d) \lhd\!\!\!+ \{s \mapsto tc\_hops(tc) + 1\}$
$act2$: $rt\_next\_hops(d) := rt\_next\_hops(d) \lhd\!\!\!+ \{s \mapsto tc\_sender(tc)\}$
$act3$: $got\_tc := got\_tc \cup \{tc\}$
$act4$: $rt\_sqn(d) := rt\_sqn(d) \lhd\!\!\!+ \{s \mapsto tc\_sqn(tc)\}$
end

We should note here that we refine event *remove_outdated_tc* by event *remove_outdated_tc1* and *remove_outdated_tc2*. Event *remove_outdated_tc1* removes an out-dated TC message from the network checking TC message sequence number, while event *remove_outdated_tc2* removes a TC message if it is not allowed to be forwarded in the network anymore (*tc_time_to_live* $<= 1$ based on the OLSR specification [9]).

Event *remove_outdated_tc1* $\widehat{=}$
extends *remove_outdated_tc*
any
$tc,\ a,\ s$
where
$grd1$: $a \in NODES$
$grd2$: $tc \in got\_tc$
$grd3$: $a \mapsto tc \in tc\_storage$
$grd4$: $tc\_sqn(tc) \leq rt\_sqn(a)(s)$
$grd5$: $s = source\_tc(tc)$
then
$act1$: $broadcast\_tc := broadcast\_tc \setminus \{tc\}$
$act2$: $tc\_storage := tc\_storage \setminus \{a \mapsto tc\}$
$act3$: $got\_tc := got\_tc \setminus \{tc\}$
end

Event *remove_outdated_tc2* $\widehat{=}$
extends *remove_outdated_tc*
any
$tc,\ a$
where
$grd1$: $a \in NODES$
$grd2$: $tc \in got\_tc$
$grd3$: $a \mapsto tc \in tc\_storage$
$grd4$: $tc\_time\_to\_live(tc) \leq 1$
then
$act1$: $broadcast\_tc := broadcast\_tc \setminus \{tc\}$
$act2$: $tc\_storage := tc\_storage \setminus \{a \mapsto tc\}$
$act3$: $got\_tc := got\_tc \setminus \{tc\}$
end

4.5 Fourth Refinement *M4*: MPR Selection

In machine *M3*, all nodes were broadcasting TC messages through the network; in this machine we restrict this by determining so called MPRs. Here, only MPR nodes are able to broadcast TC messages in the network, which helps reducing the traffic in the network. We define six new variables to model MPR selection. Every node broadcasts HELLO messages which contain the message originator (*source_hello*) as well as the one-hop neighbours of the message originator (in variable *hello_onehop_array*, shown in *inv1*) and its MPR nodes (in variable *hello_mprs*, shown in *inv6*). Variable *f_mpr_status* (*inv2*) indicates whether or not a node is an MPR. If it is true, then it is able to broadcast and forward TC messages through the network. Variable *node_one_hops* in *inv3* models the one-hop neighbours of each node. Variable *f_processed_onehop* (invariant *inv4*) is introduced to check whether or not a node has updated its one-hop neighbours (used as a guard when broadcasting HELLO messages). Nodes keep track of their MPRs, modelled by variable *node_mpr_array* in *inv5*. Invariants *inv7* and *inv8* model that only MPR nodes broadcast TC messages. In this refinement, we refine some previous events and also introduce new events.

INVARIANTS

- *inv1* : $hello\_onehop\_array \in Hello \rightarrow (NODES \rightarrow BOOL)$
- *inv2* : $f\_mpr\_status \in NODES \rightarrow BOOL$
- *inv3* : $node\_one\_hops \in NODES \rightarrow (NODES \rightarrow BOOL)$
- *inv4* : $f\_processed\_onehop \in NODES \rightarrow BOOL$
- *inv5* : $node\_mpr\_array \in NODES \rightarrow (NODES \rightarrow BOOL)$
- *inv6* : $hello\_mprs \in Hello \rightarrow (NODES \rightarrow BOOL)$
- *inv7* : $\forall s \cdot s \in NODES \wedge f\_mpr\_status(s) = FALSE \Leftrightarrow (\neg \exists tc \cdot tc \in broadcast\_tc \wedge source\_tc(tc) = s)$
- *inv8* : $\forall s \cdot s \in NODES \wedge local\_sqn(s) \neq 0 \Leftrightarrow f\_mpr\_status(s) = TRUE$

We define new event *determine_onehop_neighbours* to model selection of one-hop of neighbours of every node. When this event is executed the node determines its one-hop neighbours which is then used in HELLO messages.

Event *determine_onehop_neighbours* $\widehat{=}$

any

$s,\ d$

where

- *grd1* : $s \in NODES \wedge d \in NODES \wedge rt\_next\_hops(s)(d) = d$
- *grd2* : $rt\_hops(s)(d) = 1$
- *grd3* : $f\_processed\_onehop(s) = FALSE$

then

- *act1* : $node\_one\_hops(s) := node\_one\_hops(s) \Leftarrow \{d \mapsto TRUE\}$
- *act2* : $f\_processed\_onehop := f\_processed\_onehop \Leftarrow \{s \mapsto TRUE\}$

end

Event *broadcast_hello* is slightly modified to consider one-hop neighbours of the HELLO message originator as well as the MPR nodes of the HELLO message originator.

Event *brodcast_hello* $\widehat{=}$
extends *brodcast_hello*
any
$s,\ hello$
where
grd1 : $s \in NODES$
grd2 : $hello \in Hello$
grd3 : $hello \notin broadcast\_hello$
grd4 : $source\_hello(hello) = s$
grd5 : $s \mapsto hello \notin hello\_storage$
grd6 : $f\_processed\_onehop(s) = TRUE$
then
act1 : $broadcast\_hello := broadcast\_hello \cup \{hello\}$
act2 : $hello\_storage := hello\_storage \cup \{s \mapsto hello\}$
act3 : $hello\_mprs(hello) := node\_mpr\_array(s)$
act4 : $hello\_onehop\_array(hello) := node\_one\_hops(s)$
act5 : $f\_processed\_onehop := f\_processed\_onehop \Leftarrow \{s \mapsto FALSE\}$
end

We also refine event *receive_hello* by two events *receive_hello_onehops* and *receive_hello_update_two_hops*. We add two guards *grd4–grd5* into the event *receive_hello_onehops* stating that if the HELLO message does not have any information about the one-hop neighbours of the originator, then the actions of this event are the same as the actions in event *receive_hello*.

Event *receive_hello_onehops* $\widehat{=}$
extends *receive_hello*
any
$s,\ d,\ hello,\ a$
where
grd1 : $hello \in broadcast\_hello \setminus (got\_hello \cup deleted\_hello) \wedge source\_hello(hello) = s$
grd2 : $s \neq d$
grd3 : $d \mapsto hello \in hello\_storage$
grd4 : $a \in NODES$
grd5 : $hello\_onehop\_array(hello)(a) = FALSE$
then
act1 : $rt\_hops(d) := rt\_hops(d) \Leftarrow \{s \mapsto 1\}$
act2 : $rt\_next\_hops(d) := rt\_next\_hops(d) \Leftarrow \{s \mapsto source\_hello(hello)\}$
act3 : $got\_hello := got\_hello \cup \{hello\}$
end

Event *receive_hello_update_two_hops* is used to update the MPR nodes of the HELLO messages receiver and MPR status of the HELLO message receiver (if the receiving node is an MPR or not).

Event *receive_hello_update_two_hops* $\widehat{=}$

extends *receive_hello*

any

$s,\ d,\ hello,\ a$

where

grd1 : $hello \in broadcast\_hello \setminus (got\_hello \cup deleted\_hello) \wedge source\_hello(hello) = s$

grd2 : $s \neq d$

grd3 : $d \mapsto hello \in hello\_storage$

grd4 : $a \in NODES$

grd5 : $hello\_onehop\_array(hello)(a) = TRUE$

grd6 : $a \neq d$

then

act1 : $rt\_hops(d) := rt\_hops(d) \lhd\!\!+ \{s \mapsto 1\}$

act2 : $rt\_next\_hops(d) := rt\_next\_hops(d) \lhd\!\!+ \{s \mapsto source\_hello(hello)\}$

act3 : $got\_hello := got\_hello \cup \{hello\}$

act4 : $f\_mpr\_status := f\_mpr\_status \lhd\!\!+ \{d \mapsto TRUE\}$

act5 : $node\_mpr\_array(d) := node\_mpr\_array(d) \lhd\!\!+ \{s \mapsto TRUE\}$

end

We also modify events *broadcast_tc* and *forward_tc*. Guard *f_mpr_status(s)* = *TRUE* in event *broadcast_tc* restricts the broadcasting of TC messages to only specific nodes (MPR nodes) so that only MPRs increase their local sequence number in event *braodcast_tc*.

Event *broadcast_tc* $\widehat{=}$

extends *broadcast_tc*

any

$s,\ tc$

where

grd1 : $s \in NODES$

grd2 : $tc \in Tc$

grd3 : $tc \notin broadcast\_tc$

grd4 : $source\_tc(tc) = s$

grd5 : $s \mapsto tc \notin tc\_storage$

grd6 : $f\_mpr\_status(s) = TRUE$

then

act1 : $tc\_sender := tc\_sender \lhd\!\!+ \{tc \mapsto s\}$

act2 : $tc\_storage := tc\_storage \cup \{s \mapsto tc\}$

act3 : $broadcast\_tc := broadcast\_tc \cup \{tc\}$

act4 : $local\_sqn := local\_sqn \lhd\!\!+ \{s \mapsto local\_sqn(s) + 1\}$

end

Guard *f_mpr_status (a)* = *TRUE* in event *forward_tc* also restricts the broadcasting of TC messages by all nodes (only MPR nodes forward TC messages through network).

Event *forward_tc* $\widehat{=}$

extends *forward_tc*

any

$a,\ b,\ tc$

where

grd1 : $tc \in got\_tc \wedge source\_tc(tc) \neq a$

grd2 : $b \in NODES \land a \in NODES \land a \neq b$
grd3 : $a \mapsto tc \in tc\_storage \land b \mapsto tc \notin tc\_storage$
grd4 : $tc\_time\_to\_live(tc) > 1$
grd5 : $f\_mpr\_status(a) = TRUE$

then

act1 : $tc\_storage := (tc\_storage \setminus \{a \mapsto tc\}) \cup \{b \mapsto tc\}$
act2 : $tc\_hops := tc\_hops \lhd\!\!+ \{tc \mapsto tc\_hops(tc) + 1\}$
act3 : $tc\_sender := tc\_sender \lhd\!\!+ \{tc \mapsto a\}$
act4 : $tc\_time\_to\_live := tc\_time\_to\_live \lhd\!\!+ \{tc \mapsto tc\_time\_to\_live(tc) - 1\}$

end

5 Verification and Impact

Figuring out from which level of abstraction to start modelling and what details to add at every step of the refinement is a challenging task. We model our system in order to address modelling/proving complexity and to preserve reusability of models. In order to check if our models satisfy their correctness properties we need to prove that the invariants are preserved. In our case, correctness of packet delivery after injecting a data packet to the system, optimal route finding, and route discovery are investigated. In order to prove these, we used the Rodin platform tool to generate the proof obligations for all the models. The summary of proof statistics is displayed in Table 1 consisting of the number of proof obligations generated by Rodin platform, number of proof obligations discharged automatically and number of proof obligations discharged interactively.

Table 1: Proof Statistics

Model	Number of Proof Obligations	Automatically Discharged	Interactively Discharged
C0	4	4	0
M0	13	13	0
M1	51	47	4
C1	1	1	0
M2	102	95	7
M3	17	16	1
M4	34	23	11
Total	222	199	23

In addition to proving certain properties that hold for the OLSR protocol, our main contribution consists in the unpacking of OLSR complexity into five abstraction layers, as illustrated in Fig. 1. The first two layers describe the abstract behaviour of a routing protocol; the following two add the needed infrastructure to model proactive routing behaviour; and the last layer only introduces the specifics of OLSR, that of MPR-based working. Hence, our modelling is highly reusable: routing protocols can be developed based on the first two abstraction layers, even reactive routing protocols; other proactive routing protocols can be developed based on abstraction layers three and four, for instance the BATMAN routing protocol. Hence, we have presented a case study that emphasises the power of

refinement-based methods as well as their reusability and adaptability. Reusing our models *M0 – M3* in modelling other routing protocols is left for future research.

6 Related Work

Formal specification and analysis techniques have been used for investigating the correctness and performance of different wireless networks and their routing protocols such as OLSR, DSR, etc. In the following, we review samples related to model checking and theorem proving.

In [13], we used model checking techniques, namely Uppaal [7], to model, analyse and verify the OLSR protocol for different properties, e.g., route establishment, packet delivery, route optimality, etc, based on the OLSR specification [9]. Our verification was performed for static and dynamic network topologies up to 5 nodes. Our investigation of the OLSR behaviour was performed for small networks; nevertheless, we were able to report some problems of the protocol and to sketch some modifications [14] to fix these problems and to improve the performance of OLSR. In our current paper, the analysis scale is significantly different: by applying theorem proving techniques, we are not restricted by the number of nodes and can verify our system for realistic, large-scale networks.

Steele and Andel [16] analyse the OLSR protocol using the model checker Spin [11]. They model OLSR and use Linear Temporal Logic (LTL) in order to investigate the correct functionality of OLSR. They verify their model for different properties such as route discovery, MPR selection, etc. Their analysis is limited to networks with at most 4 nodes due to the state space explosion of SPIN. When taking symmetries into account they analyse 17 topologies.

Kamali et al. [12] apply refinement techniques to model and analyse wireless sensor-actor networks. They proved that failed actor links can be temporarily replaced by communication over the sensor infrastructure, when certain assumptions hold. They employ the Event-B formalisation (based on theorem proving) and use the Rodin tool to carry out their proofs. While there is a strong similarity between the nature of distributed sensor-based recovery and that of distributed OLSR, the focus in their study is different.

Méry and Singh [15] present a stepwise development of the DSR protocol using Event-B to analyse and reason about the behaviour of this protocol. They refine their abstract model in 5 refinement steps and define invariants to verify their system for safety and liveness properties. These properties are established by proof of defined invariants, refinement of events, etc, using the Rodin platform. It is interesting to note that our initial models are quite similar, as both start from the basics of routing protocols. However, while DSR is a reactive protocol, OLSR is proactive, so our developments start to differ significantly from our third layer on.

Somewhat older related work [10] discusses the modelling of topology discovery in Event-B. This is quite interesting, as proactive behaviour does, in fact, (continuously) construct for each node a certain view of the rest of the network. When comparing our approach to the topology discovery [10], we observe ours to be developed more from the routing protocol's perspective and hence more practical: we can use it to analyse the properties of a specific protocol with it. Nevertheless, theirs [10] is a very elegant conceptual development, taking explicitly into account the failed and non-failed links, safety properties, as well as liveness properties defining the notion of a stable system (one where the view of the network in the model is the accurate one). In our paper, we abstract away from failed links and assume that the protocol continuously learns about links until they are all discovered. At that point we would have a stable system, but we do not focus on it here. In fact, in future work we aim at also modelling the failing of links (as in WMNs this continuously happens) and build our protocol so that it continuously verifies that it has the newest information on its connectivity. However, that will not be a stable system, as links may go up and down continuously, it would be just a best effort of topology discovery. The fact that the topology discovery [10] is developed some years ago is visible in their ratio of automatic vs interactive proofs: roughly half of their proofs are interactive, probably due in large part to the preliminary version of the Rodin platform at the time.

7 Conclusions

Creating a model in Event-B for analysing the OLSR protocol was originally motivated by our previous work on creating an Uppaal model [13] for this protocol. While we could experiment with Uppaal on various properties, the main drawback of the Uppaal model was its limitation to 5-node topologies. We needed to understand whether OLSR works well for arbitrary topologies and thus, we created the Event-B model in this paper. This model also has some limitations, such as abstracting away from the timing behaviour and not including the deletion (failure) of links. We believe that the former drawback simply needs a more involved modelling approach, as we explain below. A deeper modelling approach is also needed for addressing the latter drawback, with the additional comment that in Uppaal we have studied what happens if a particular link in a particular 5-node topology fails. Hence, while the current Event-B model is not complete yet, we believe it demonstrates several achievements.

In this paper, we have modelled the correct behaviour of OLSR with respect to finding routes to all destinations and delivering data packets along these routes. We have also showed that OLSR does not find optimal routes for all the destinations. These properties hold true also for our previous Uppaal model. However, the greatest advantage of the Event-B model consists in generality and reusability. Our proved management of OLSR complexity is a central contribution. The for-

mal development of the protocol is carried out in several layers: routing protocol, proactive behaviour protocol, and OLSR. These layers are reusable and adaptable, as argued in Section 5. The last layer elegantly shows how OLSR is a refinement of proactive routing, delivering a more efficient algorithm: only the selected MPR nodes are enabled to send and forward control messages, so as not to flood the network.

We plan to continue the research reported in this paper in several directions. First, we are working on the finishing touches of a companion paper where we compare the Uppaal model with the Event-B model for OLSR; in that paper we also discuss the relative advantages of using Uppaal and Event-B for analysing protocols. Second, there is timing behaviour in OLSR [9] that we abstracted away in this paper. By including it though, we would be able to reason about timing properties of OLSR, hence we plan to employ an approach for time modelling in Event-B, such as Hybrid Event-B [6] or [4] for instance. This would imply that all variables except clocks are functions of time, so a slight change of perspective is needed here. Third, there is a remarkable resemblance between the basic behaviour of data packets and the other control messages in the network: they are all sent, received, locally stored. Hence, we plan to investigate a theory of messages in connection with routing protocols in Event-B, much in the spirit of other theories [8] already introduced in the Rodin platform. This would increase the reusability of both the proposed models *and* proofs. Modelling adding *and* removal of links in the WMNs would make our models more realistic. Finally, showing how to reuse various models for other routing protocols would clearly demonstrate the advantages of Event-B and the refinement approach.

References

[1] Abrial, J.R.: The B-book: Assigning Programs to Meanings. Cambridge University Press (1996)

[2] Abrial, J.R.: Modeling in Event-B - System and Software Engineering. Cambridge University Press (2010)

[3] Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. STTT 12(6), 447–466 (2010)

[4] Abrial, J.R., Su, W., Zhu, H.: Formalizing hybrid systems with event-b. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) Abstract State Machines, Alloy, B, VDM, and Z, Lecture Notes in Computer Science, vol. 7316, pp. 178–193. Springer (2012)

[5] Back, R.J., Sere, K.: From action systems to modular systems. In: Formal Methods Europe ?94: Industrial Benefit of Formal Methods. Lecture Notes in Computer Science, vol. 873, pp. 1–25. Barcelona, Spain (1994)

[6] Banach, R., Butler, M., Qin, S., Verma, N., Zhu, H.: Core hybrid event-b i: Single hybrid event-b machines. Science of Computer Programming 105, 92 – 123 (2015)

[7] 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)

[8] Butler, M., Maamria, I.: Practical theory extension in event-b. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) Theories of Programming and Formal Methods, Lecture Notes in Computer Science, vol. 8051, pp. 67–81. Springer (2013)

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

[10] Hoang, T.S., Kuruma, H., Basin, D., Abrial, J.R.: Developing topology discovery in event-b. In: Leuschel, M., Wehrheim, H. (eds.) Integrated Formal Methods, Lecture Notes in Computer Science, vol. 5423, pp. 1–19. Springer (2009)

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

[12] 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)

[13] Kamali, M., Höfner, P., Kamali, M., Petre, L.: Formal analysis of proactive, distributed routing. In: 13th International Conference on Software Engineering and Formal Methods (SEFM 2015). vol. 9276, pp. 175–189. Springer (2015)

[14] Kamali, M., Petre, L.: Improved recovery for proactive, distributed routing. In: 20th International Conference on Engineering of Complex Computer Systems (ICECCS 2015). pp. 178–181. IEEE (2015)

[15] Méry, D., Singh, N.K.: Analysis of DSR protocol in Event-B. In: SSS. Lecture Notes in Computer Science, vol. 6976, pp. 401–415. Springer (2011)

[16] 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)

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3-5 A, 20520 TURKU, Finland | www.tucs.fi

University of Turku
Faculty of Mathematics and Natural Sciences
- Department of Information Technology
- Department of Mathematics

Turku School of Economics
- Institute of Information Systems Sciences

Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

ISBN 978-952-12-3350-0
ISSN 1239-1891