Analysis of DSR Protocol in Event-B

Dominique Méry and Neeraj Kumar Singh

Université Henri Poincaré Nancy 1
LORIA, BP 239, 54506 Vandoeuvre lès Nancy, France
{mery,singhnne}@loria.fr

Abstract. This paper presents an incremental formal development of the Dynamic Source Routing (DSR) protocol in Event-B. DSR is a reactive routing protocol, which finds a route for a destination on demand, whenever communication is needed. Route discovery is an important task of any routing algorithm and formal specification of it, itself is a challenging problem. The specification is performed in a stepwise manner composing more advanced routing components between the abstract specification and topology. It is verified through a series of refinements. The specification includes safety properties as set of invariants, and liveness properties that characterize when the system reaches stable states. We establish these properties by proof of invariants, event refinement and deadlock freedom. The consequence of this incremental approach helps to achieve a high degree of automatic proof. Our approach can be useful for formalizing and developing other kinds of reactive routing protocols (i.e. AODV etc.).

Keywords: Abstract model, Event-B, Event-driven approach, Proof-based development, Refinement, Ad hoc Network.

1 Introduction

Formal models have a valuable role to play in validating requirements and designs for distributed systems. In a mobile ad hoc networks, nodes move arbitrarily and change the network topology. Frequently changing topology presents a fundamental challenge for routing protocols. This paper presents a case study on the Dynamic Source Routing (DSR) protocol [1]. Reactive routing protocol is generally not dependent on exchanges of periodic route information and route calculations. Instead, whenever a route is needed the node has to perform a route discovery before it can send any packet to a destination node. Our approach is here to specify and formally develop the DSR protocol. We use an incremental development of the DSR protocol with stepwise refinements in Event-B [2,3]. Event-B [2,3] is a formal modeling language, which supports refinement based formal development. We proceed by constructing the proof-based series of models, where the initial model specifies the system requirements and final model describes the resulting system.

It is a significant case study in specifying and developing the real routing protocol algorithms. In routing protocols each host works as a router and constructs a graph representing the network topology. In this graph, vertices and edges represent routing nodes and direct connection between nodes, respectively. Each node uses this graph to find the optimized routing table and determines the correct route from source node to destination node. The main challenging task in route discovery is to find the exact distribution of nodes in a dynamic network and routing updates after changing network topology.

To specify the correct desired properties of protocol at abstract level and in carrying out the development and proofs in subsequent refinement models, is a challenging problem. This challenging problem comes from the fact that the protocol should function in dynamically changing environment. The main characteristics of an ad hoc network is dynamic behavior of the network: nodes can be added and deleted in a dynamic manner. The topology information in all the reactive protocol is only transmitted by nodes on-demand such as a node wishes to transmit the data packets to a node to which it has no route, it will generate a route request message that will be flooded in a limited way to other nodes. A route is considered found when the route request message reaches either at a destination itself, or at an intermediate node with a valid route entry for the destination node.

One of the key aspect of our development is to verify $stability$ of the system. $Stability$ of the system is a most important property of this chaotic networks which implies correct local view of the current system. Intuitively, in stable states, all nodes have the maximum knowledge of the environment that can be acquired by route discovery and communication. This notion of system stability is an instance of the general notion of a *stable system property* [4].

The models of DSR protocol must be validated to ensure that they meet the requirements. Our abstract specification includes events of basic communication protocol. The nature of the refinement that we verify using Rodin [2] proof tools are safety refinement. Thus the behavior of final resulting system is preserved by abstract model as well as in correctly refined models. Proof-based development methods [3] integrate formal proof techniques in the development of software systems. The main idea is to start modeling with an abstract model and details are gradually added to the abstract model to produce a sequence of concrete events. The relationship between two successive models is known as *refinement* [3]. The current work intends to explore problems related to the modeling of distributed systems where an environment is changing dynamically. Moreover, the stepwise development of the DSR protocol model helps to discover the exact behavior of basic communication protocol and route discovery protocol in dynamic environment.

The outline of remaining paper organizes as follows. Section 2 describes the modeling framework, which outline some general idea of modeling, that we found useful in this work. In section 3, we describe an informal description of the DSR protocol. Requirements and assumptions are described in section 4. Section 5 explores the formal development of the DSR protocol using stepwise refinement. Finally, section 6 presents discussion and conclusion of the work.

2 The Modeling Framework

We will summarize the concepts of the Event-B modeling language developed by Abrial [5,3] and will indicate the links with the tool called RODIN [2]. Considering the Event-B modeling language, we notice that the language can express *safety* properties, which are either *invariants* or *theorems* in a machine corresponding to the system. Recall that two main structures are available in Event-B:

– Contexts express static information about the model.
– Machines express dynamic information about the model, invariants, safety properties, and events.

An Event-B model defines either a context or a machine. A machine organizes events modifying state variables and it uses static information defined in a context. These basic structure mechanisms are extended by the refinement mechanism which provides a mechanism for relating an abstract model and a concrete model by adding new events or by adding new variables. This mechanism allows us to develop gradually Event-B models and to validate each decision step using the proof tool. The refinement relationship should be expressed as follows: a model M is refined by a model P, when P is executing M. The final concrete model is close to the behavior of real system that executes events using real source code. We give details now on the definition of events, refinement and guidelines for developing complex system models.

2.1 Modeling Actions over States

The event-driven approach [5,3] is based on the B notation. It extends the methodological scope of basic concepts to take into account the idea of *formal models*. Briefly, a formal model is characterized by a (finite) list x of *state variables* possibly modified by a (finite) list of *events*, where an invariant $I(x)$ states properties that must always be satisfied by the variables x and *maintained* by the activation of the events. In the following, we summarize definitions and principles of formal models and explain how they can be managed by tools [2].

Generalized substitutions are borrowed from the B notation. They provide a means to express changes to state variable values. In its general form, an event has three main parts, namely a list of local parameters, a guard and a relation over values denotes pre values of variables and post values of variables. The most common event representation is (ANY t WHERE $G(t,x)$ THEN $x :| (R(x,x',t))$ END). The *before–after* predicate $BA(e)(x,x')$, associated with each event, describes the event as a logical predicate expressing the relationship linking the values of the state variables just before (x) and just after (x') the *execution* of event e. The form is semantically equivalent to $\exists t\cdot (G(t,x) \wedge R(x,x',t))$.

Table 1. Event-B proof obligations

PROOF OBLIGATIONS

- (INV 1) $Init(x) \Rightarrow I(x)$
- (INV 2) $I(x) \wedge BA(e)(x,x') \Rightarrow I(x')$
- (FIS) $I(x) \wedge \mathsf{grd}(e)(x) \Rightarrow \exists y.BA(e)(x,y)$

Proof obligations (INV 1 and INV 2) are produced by the Rodin tool [2] from events to state that an invariant condition $I(x)$ is preserved. Their general form follows immediately from the definition of the before–after predicate $BA(e)(x,x')$ of each event e (see Table-2). Note that it follows from the two guarded forms of the events that this obligation is trivially discharged when the guard of the event is false. Whenever this is the case, the event is said to be *disabled*. The proof obligation FIS expresses the feasibility of the event e with respect to the invariant I.

2.2 Model Refinement

The refinement of a formal model allows us to enrich the model via a *step-by-step* approach and is the foundation of our correct-by-construction approach [6]. Refinement provides a way to strengthen invariants and to add details to a model. It is also used to transform an abstract model to a more concrete version by modifying the state description. This is done by extending the list of state variables (possibly suppressing some of them), by refining each abstract event to a corresponding concrete version, and by adding new events. The abstract (x) and concrete (y) state variables are linked by a *gluing invariant* $J(x, y)$. A number of proof obligations make sure that (1) each abstract event is correctly refined by its corresponding concrete version, (2) each new event refines $skip$, (3) no new event takes control for ever, and (4) relative deadlock freedom is preserved. Details of the formulation of these proofs follows.

We suppose that an abstract model AM with variables x and invariant $I(x)$ is refined by a concrete model CM with variables y and gluing invariant $J(x, y)$. If $BA(e)(x, x')$ and $BA(f)(y, y')$ are the abstract and concrete before–after predicates of the same event, e and f respectively, we have to prove the following statement, corresponding to proof obligation (1):

$$\boxed{I(x) \ \wedge \ J(x,y) \ \wedge \ BA(f)(y,y') \ \Rightarrow \ \exists x' \cdot (BA(e)(x,x') \ \wedge \ J(x',y'))}$$

Now, proof obligation (2) states that $BA(f)(y, y')$ must refine $skip$ ($x' = x$), generating the following simple statement to prove (2).

$$\boxed{I(x) \ \wedge \ J(x,y) \ \wedge \ BA(f)(y,y') \ \Rightarrow \ J(x,y')}$$

In refining a model, an existing event can be refined by strengthening the guard and/or the before–after predicate (effectively reducing the degree of nondeterminism), or a new event can be added to refine the *skip* event. The feasibility condition is crucial to avoiding possible states that have no successor, such as division by zero. Furthermore, this refinement guarantees that the set of traces of the refined model contains (up to stuttering) the traces of the resulting model. The refinement of an event e by an event f means that the event f simulates the event e.

The Event-B modeling language is supported by the Rodin platform [2] and has been introduced in publications [3,5], where the many case studies and discussions about the language itself and the foundations of the Event-B approach. The language of *generalized substitutions* is very rich, enabling the expression of any relation between states in a set-theoretical context. The expressive power of the language leads to a requirement for help in writing relational specifications, which is why we should provide guidelines for assisting the development of Event-B models.

3 Informal Description of DSR Protocol

The DSR protocol is a simple and efficient routing protocol designed specifically to use in multi-hop wireless ad hoc networks of mobile nodes. It allows the network to be completely self-organizing and self-configuring, without the need for any existing network infrastructure or administration. In source routing techniques, a sender determines the complete sequence of nodes through which it forwards the data packet. The sender explicitly lists this route in the packets header, identifying each forwarding 'hop' by the address of the next node to which transmits the data packet on its way to the destination node. The sender then transmits the packet over its wireless network interface to the first hop identified in the source route. When a host receives a packet, if this host is not the destination of the packet, it simply transmits the packet to the next hop identified in the source route in the packet header. Once the packet reaches its destination, the packet is delivered to the host. The protocol presented here is explicitly designed for use in the wireless environment of an ad hoc network. There are no periodic router advertisements in the protocol. Instead, when a node needs a route to another node, it dynamically determines one based on a local routing table or a route cached information and on the results of a route discovery protocol [1]. DSR consists of two mechanisms: $route\ discovery$ and $route\ maintenance$.

Route Discovery: Whenever a source needs to communicate to a destination and does not have a route in its routing table, it broadcasts a route request (RREQ) message to find a route. Each neighbor receives the RREQ and (if it has not already processed the same request earlier) appends its own address to the address list in the RREQ and rebroadcasts the packet. This process continues until either the maximum hop counter is exceeded (and RREQ is discarded) or the destination is reached. In the latter case, the destination receives the RREQ, appends its address and generates a route reply packet (RREP) back towards the source using the reverse of the accumulated route [1].

Route Maintenance: Route maintenance is used to manage (cache, expire, switch among) previously discovered routes. Each node along the route, when transmitting the packet to the next hop, is responsible for detecting next connected link. When the retransmission and acknowledgement mechanism detects that the link is broken, the detecting node returns a route error packet (RERRP) to the source of the packet. The node will then search its route cache to find if there is an alternative route to the destination of this packet. If there is one, the node will change the source route in the packet header and send it using this new route. When a route error packet (RERRP) is received or overheard, the link in error is removed from the local route cache, and all routes which contain this hop must be truncated at that point [1]. The source can then attempt to use any other route to the destination that is already in its route cache, or can invoke route discovery again to find a new route.

4 Requirements and Assumptions

The protocol must work in an environment where the status of links may change at any time. If the environment changes sufficiently rapidly, then links reported as down may actually be up and vice versa. Hence the local routing table may bear little relationship to the actual network topology. To tackle this problem, we focus on the limiting, and most important, case of the algorithm's behavior: its behavior when the environment is sufficiently quiescent. In this case, we expect that the local routing table will eventually "stabilize" to states of the actual global topology. According to the basic graph theory, any graph can be decomposed into a collection of strongly-connected components. Our main system requirements are:

System Requirement 1: Data packet must be transmitted successfully from source node to destination node in a dynamic ad hoc network.

System Requirement 2: If the environment is inactive for a sufficiently long time then communication stabilizes and each node has the correct view of the links between all nodes in its connected subnetwork.

System Requirement 3: Route discovery protocol must discover a new route from the connected network where the status of links may change at any time.

Before developing the formal model of DSR protocol, we have some assumptions as follows:

- There are finite numbers of nodes or hosts.
- There are directed, one-way links between some pairs of distinct nodes. Links may come up and go down at any time.
- Nodes are communicating by broadcasting where node (x) sends a message to other node (y) when they are directly connected.
- When a link goes down, any message sent on it and not yet received are lost. This reflects that communication is asynchronous. There is a delay between message transmission and reception, and messages can be lost during this time interval.
- The hosts do not continuously move so rapidly as to make the flooding of every packet.

5 Formal Development

DSR protocol development is expressed in an abstract and general way. We describe the incremental development of DSR protocol in two phases as basic communication protocol and route discovery protocol. We develop the six models related by refinements. The initial model formalizes our system requirements and environmental assumptions, whereas the subsequent models introduce design decisions for the resulting system.

Initial Model : To specify basic communication protocol of data packet sending, receiving, losing, and network topology changes using some initial events (*sending, receiving, losing, remove_link* and *add_link*).

Refinement 1 : Introducing store and forward architecture for data packets passing from source node to destination node.
Refinement 2 : Introducing local routing table.
Refinement 3 : Introducing route discovery protocol to discover a new route.
Refinement 4 : Provides more detail information about route discovery protocol.
Refinement 5 : Introducing sequence numbers for tracking fresh route request packets information.

5.1 The Context and Initial Model

We define a carrier set ND of network nodes. It is finite and is represented by an axiom $(axm1)$. The network is supported by a directed graph g built on ND, is defined by an axiom $(axm2)$. An axiom $(axm3)$ specifies that there is no self loop connection in the network means any node is not directly connected to itself. Axioms $(axm4$ and $axm5)$ represent that the total functions map from a carrier set Msg to a set of nodes ND. The constants $source$ and $target$ are two necessary fields of a data packet for presenting source and destination references. An additional constant $(closure)$ is defined by axioms$(axm6 - axm9)$ that formalizes the transitive closure of binary relations between a set of nodes (ND). Note that ";" denotes forward relational composition.

$axm1 : finite(ND)$	$axm6 : closure \in (ND \leftrightarrow ND) \rightarrow (ND \leftrightarrow ND)$
$axm2 : g \subseteq ND \times ND$	$axm7 : \forall r \cdot r \subseteq closure(r)$
$axm3 : id(ND) \cap g = \varnothing$	$axm8 : \forall r \cdot closure(r); r \subseteq closure(r)$
$axm4 : source \in Msg \rightarrow ND$	$axm9 : \forall r, s \cdot r \subseteq s \wedge s; r \subseteq s \Rightarrow closure(r) \subseteq s$
$axm5 : target \in Msg \rightarrow ND$	

In abstract model, we formalize behavior of the basic communication protocol and dynamic environment where links may go ups and down at any time. New variables $sent$, got and $lost$ $(inv1 - inv3)$ are introduced to represent the set of sending data packets by any source node, successfully received data packets by any destination node and lost data packets due to network failure, respectively. A variable $ALinks$ (i.e active link) represents a set of links that currently up and keeps up-to-date information about all adding and removing links in the network. An invariant $(inv5)$ represents a safety property and states that all the received and lost data packets are subset of the sending data packets.

The sets got and $lost$ are disjoint $(inv6)$ since a data packet cannot be simultaneously both received and lost. We include events modeling, atomic transfer of the data packets between moving nodes, successfully receiving of the data packets by destination node, losing of the data packets due to network failure and dynamic changing in network topology.

$inv1 : sent \subseteq Msg$
$inv2 : got \subseteq Msg$
$inv3 : lost \subseteq Msg$
$inv4 : ALinks \in ND \leftrightarrow ND$
$inv5 : got \cup lost \subseteq sent$
$inv6 : got \cap lost = \varnothing$

```
EVENT sending
  ANY   s,t,data_msg
  WHERE
    grd1 : $data\_msg \in Msg$
    grd2 : $data\_msg \notin sent$
    grd3 : $s \in ND \wedge t \in ND \wedge s \neq t$
    grd4 : $source(data\_msg) = s$
    grd5 : $target(data\_msg) = t$
  THEN
    act1 : $sent := sent \cup \{data\_msg\}$
  END
```

There are five significant events in our abstract model. An event $sending$ represents the sending of a data packet $(data\_msg)$ from a source node (s) to a destination node (t). Guards of this event state that a new data packet $(data\_msg)$ is sending from the source node (s) to the destination node (t) and both source and destination are different nodes. An event $receiving$ represents for successful receiving of the data packet $(data\_msg)$ by the destination node (t). A guard $(grd1)$ of $receiving$ states that the sending data packet $(data\_msg)$ is a member of $sent$ and the data packet is not received by either got or $lost$ variables. The data packet $(data\_msg)$ has correct references of the source node (s) and the destination node (t) is represented by a guard $(grd2)$.

```
EVENT receiving
  ANY  s,t,data_msg
  WHERE
    grd1 : $data\_msg \in sent \setminus (got \cup lost)$
    grd2 : $source(data\_msg) = s \wedge$
           $target(data\_msg) = t$
  THEN
    act1 : $got := got \cup \{data\_msg\}$
  END
```

```
EVENT losing
  ANY  s,t,data_msg
  WHERE
    grd1 : $data\_msg \in sent \setminus (got \cup lost)$
    grd2 : $source(data\_msg) = s \wedge$
           $target(data\_msg) = t$
    grd3 : $s \mapsto t \notin closure(ALinks)$
  THEN
    act1 : $lost := lost \cup \{data\_msg\}$
  END
```

An event $losing$ represents loss of data packets due to network failure or suddenly powered off of any node or moving of node to new location, and disconnected from the network. Guards state that the sending data packet $(data\_msg)$ is not received by either got or $lost$ variables and there is not any valid connected route from the source node (s) to the destination node (t). Guard $grd3$ states that a data packet never gets the destination (t) node when path is broken.

```
EVENT add_link
  ANY  x,y
  WHERE
    grd1 : $x \mapsto y \notin ALinks$
    grd2 : $x \neq y$
  THEN
    act1 : $ALinks := ALinks \cup \{x \mapsto y\}$
  END
```

```
EVENT remove_link
  ANY  x,y
  WHERE
    grd1 : $x \mapsto y \in ALinks$
    grd2 : $x \neq y$
  THEN
    act1 : $ALinks := ALinks \setminus \{x \mapsto y\}$
  END
```

There is no more fixed infrastructure in wireless ad hoc network and every node in the network works as router and all nodes move from one place to other place without giving any information, so network link information always changes. For modeling this dynamic behavior in the system we have proposed the two events add_link and $remove\_link$. Some new arbitrary links come up and some old links are removed from the network. New links are added to the set of $ALinks$ and old link are removed from the set $ALinks$ (if it is not existing). This event always keeping up-to-date information of the ad hoc network.

5.2 First Refinement : Store and Forward Architecture

In the abstract model, we have presented that the data packets have been transferred in an atomic step from the source node to the destination node. But in real protocol the data packet is transferred hop by hop from the source node (s) to the destination node (t). So our goal is to model the store and forward architecture, where all nodes are not directly connected, and a data packet must pass through a number of intermediate nodes before reaching to the destination node. We introduce a new variable $gstore$ as binary relation between ND and Msg is represented by invariant ($inv1$).

$$
\begin{array}{l}
inv1 : gstore \in ND \leftrightarrow Msg \\
inv2 : \forall i \cdot i \in ND \wedge i \in dom(gstore) \Rightarrow (got \cup lost) \cap gstore[\{i\}] = \varnothing \\
inv3 : ran(gstore) \cup (got \cup lost) = sent \\
inv4 : \forall i \cdot i \in ND \Rightarrow gstore[\{i\}] \subseteq sent \\
inv5 : \forall m \cdot m \in Msg \wedge m \notin sent \Rightarrow \begin{pmatrix} m \notin got \quad \wedge m \notin lost \quad \wedge \\ (\forall i \cdot i \in ND \Rightarrow i \mapsto m \notin gstore) \end{pmatrix} \\
inv6 : \forall m, i, j \cdot i \mapsto m \in gstore \wedge j \mapsto m \in gstore \Rightarrow i = j
\end{array}
$$

In the network, any data packet is stored by either $got \cup lost$ or in local variable $gstore$ by any node is represented by invariant ($inv2$). Invariant ($inv3$) represents a set of total distributed data packets $(ran(gstore) \cup (got \cup lost))$ in the network is equal to the sending data packets ($sent$). Each sending data packet is belonging from the set of sending data packets ($sent$) is given in invariant ($inv4$). Next invariant ($inv5$) states that a new data packet is not a member of the network distributed data packets if it is not member of the sending data packets ($sent$). Same data packet is not mapped by two different nodes in relation $(gstore)$ is represented by last invariant $(inv6)$, means a node cannot store contradictory information about the same data packet.

A new event $forward$ introduces in this refinement, is used to transfer the data packets between two connected neighbouring nodes in the route. First two guards represent that a new sending data packet is not received by $got \cup lost$, and intermediate nodes x and y are directly connected. Third and fourth guards state that a destination node is t of a data packet ($data\_msg$), and an intermediate node x is not the destination node (t). Last two guards represent that the data packet ($data\_msg$) is stored at the node x, not at the node y. In this refinement, we introduce some new guards and actions in events $sending$, $receiving$ and $losing$.[1]

```
EVENT forward
  ANY   t,x,y,data_msg
  WHERE
    grd1 : $data\_msg \in sent \setminus (got \cup lost)$
    grd2 : $x \mapsto y \in ALinks$
    grd3 : $target(data\_msg) = t$
    grd4 : $x \neq target(data\_msg)$
    grd5 : $x \mapsto data\_msg \in gstore$
    grd6 : $y \mapsto data\_msg \notin gstore$
  THEN
    act1 : $gstore := (gstore \setminus \{x \mapsto data\_msg\}) \cup$
           $\{y \mapsto data\_msg\}$
  END
```

```
EVENT sending
  $\oplus$ $grd7 : s \mapsto data\_msg \notin gstore$
  $\oplus$ $act2 : gstore := gstore \cup$
                 $\{s \mapsto data\_msg\}$
EVENT receiving
  $\oplus$ $grd4 : t \mapsto data\_msg \in gstore$
  $\oplus$ $act2 : gstore := gstore \setminus$
                 $\{t \mapsto data\_msg\}$
EVENT losing
  $\oplus$ $grd4 : x \mapsto data\_msg \in gstore$
  $\oplus$ $act2 : gstore := gstore \setminus$
                 $\{x \mapsto data\_msg\}$
```

Note that, together with the events $sending$, $receiving$, $losing$, $remove\_link$, add_link and $forward$ from initial model and all defined invariants establish **System Requirement 1**.

5.3 Second Refinement : Routing Update

In this refinement, we introduce a routing table or a route cache for updating the route information from the dynamic changing network. In the DSR protocol any node updates the local routing table, when a node wants to send data packets to any destination node and a route is not available in a local routing table. We define a new variable $alinks$ as $alinks \in ND \rightarrow (ND \leftrightarrow ND)$, and it represents that the routing information is stored by each node. The local routing table ($alinks$) always keeps some stale links information due to continue changing of nodes location in the ad hoc network. We introduce a new event $update\_routing\_table$ for updating the routing table. First two guards of this event represent that the path is not existing between a source node (s) to a destination node (t). A set of links, which generates a route from the source node (s) to any other node (x) is represented as a strongly connected graph by last three guards ($grd3 - grd5$). An action

[1] $\oplus$: To add a new guard and an action. , $\ominus$: To remove an old guard and an action

$(act1)$ of $update\_routing\_table$ states that the set of nodes E updates their local routing table using a variable ($routeSet$). We introduce local routing table variable $alinks$ and some new guards in all other events of basic communication protocol.

```
EVENT update_routing_table
  ANY   s,t,E,routeSet
  WHERE
    grd1 : $s \in ND \wedge t \in ND$
    grd2 : $s \mapsto t \notin closure(alinks(s))$
    grd3 : $routeSet \in ND \leftrightarrow ND$
    grd4 : $E \subseteq \{x | x \in ND \wedge$
           $s \mapsto x \in closure(ALinks)\}$
    grd5 : $routeSet \subseteq closure(E \lhd ALinks)$
  THEN
    act1 : $alinks := alinks \mathbin{<\!\!+} (\lambda n \cdot n \in E | alinks(n) \cup$
           $routeSet)$
  END
```

```
EVENT sending
   $\oplus$ $grd8 : s \mapsto t \in closure(alinks(s))$
   $\oplus$ $act2 : gstore := gstore \cup$
             $\{s \mapsto data\_msg\}$
EVENT losing
   $\ominus$ $grd3 : s \mapsto t \notin closure(ALinks)$
   $\oplus$ $grd5 : s \mapsto t \notin closure(alinks(x))$
EVENT forwarding
   $\oplus$ $grd7 : y \mapsto t \in closure(alinks(x))$
```

One of the key aspects of our development strategy is to specify a so-called *observer event* [4]. This event ($stabilize$) has no effect on this system state itself as its action is **skip**. Rather, its guard is used to define the notion of a *stable state* of the system.

```
EVENT stabilize
  ANY
  WHERE
    grd1 : $\forall x, y \cdot x \mapsto y \in ALinks \Leftrightarrow x \mapsto y \in alinks(y)$
    grd2 : $\forall n, m \cdot m \mapsto n \in closure(ALinks) \Rightarrow$
           $(\forall k \cdot (k \mapsto m \in alinks(n) \Leftrightarrow k \mapsto m \in alinks(m)))$
  THEN
    skip
  END
```

First guard of event $stabilize$ represents that every node y knows the correct status of all connected links, i.e., y has detected all environment changes with respect to connected links. The next guard represents that if there is a path from a node m to n, then n has the same (up) information as m for all connected links to m. Hence, the observer event fires in those states where nodes know the correct status of their neighbors and this status has already been propagated through the network along all links. Intuitively, in stable states, all nodes have the maximum knowledge of the environment that can be acquired by route discovery and communication. We say that the system is in stable state when observer event ($stabilize$) can fire.[2] A central property that we proved is as follows:

Theorem 1 (Stability implies correct local view). *If the system is stable, then for any strongly-connected component G in the network and any node n in G, n has the correct view of the status (up) of all links in G.*

We formulate this theorem in Event-B as follows, where $guardStablize$ refers to the guards of the observe event ($stabilize$).

$$
\begin{array}{l}
guardStablize \\
\quad \Rightarrow (\forall G \cdot (\forall f, l \cdot f \in G \wedge l \in G \wedge f \neq l \Rightarrow f \mapsto l \in closure(ALinks)) \\
\qquad \Rightarrow (\forall n \cdot n \in G \\
\qquad\quad \Rightarrow G \lhd alinks(n) \rhd G = G \lhd ALinks \rhd G)
\end{array}
$$

Here, a set of nodes G defines a strongly-connected component of the graph whose edge relation defines by $ALinks$, when for every distinct pair of node f and l in G, then $f \mapsto l \in closure(ALinks)$. The operators $\lhd$ and $\rhd$ respectively restrict the domain

[2] This notion of system stability is an instance of the general notion of a *stable system property* [7,4],which is a property P is true of any reachable state s then P is true of all states reachable from s.

and the range of relation to a set. The theorem itself constitutes part of the proof of **System Requirements 2**. Namely, in a stable state, each node has the correct view of all links in its strongly-connected components.

5.4 Third Refinement : Route Discovery Protocol

The route discovery protocol is an important and complex refinement of this model. We define two carrier sets rrq and rrp of route request packets and route reply packets, respectively. Two extra constants $source\_rrq$, and $target\_rrq$ represent the total function maps a set of route request packets rrq to a set of nodes ND, for storing the source and destination references in each route request packet. A new constant ($source\_rrp$) represents the total function maps a set of route reply packets rrp to a set of nodes ND for initializing the source node for each route reply packet. Two new variables $bcast\_rrq$ and $network\_rrp$ are defined as a subset of route request packets (rrq) and route reply packets (rrp), respectively.

```
$axm1 : source\_rrq \in rrq \rightarrow ND$
$axm2 : target\_rrq \in rrq \rightarrow ND$
$axm3 : source\_rrp \in rrp \rightarrow ND$
$inv1 : bcast\_rrq \subseteq rrq$
$inv2 : network\_rrp \subseteq rrp$
```

The route request packet identifies the node, referred to as the destination node of the route discovery, for which route is requested. If the route discovery is successful then the source node receives a route reply packet listing a sequence of network hops through which it may reach to the destination node.

Two new events $broadcast\_rrq$ and $received\_rrq$ are introduced in this refinement of the route discovery protocol. The event $broadcast\_rrq$ broadcasts a route request packet for discovering a route to any destination node. First two guards ($grd1 - grd2$) of this event represent that the route is not existing between the source node (s) to the destination node (t). Next guard presents type of rrq_pkt. Last three guards ($grd4 - grd6$) state that each new route request packet rrq_pkt have references of the source node (s) and the destination node (t), then the route request packet rrq_pkt is broadcasted by initial node for discovering a new route.

```
EVENT broadcast_rrq
  ANY    s, t, rrq_pkt
  WHERE
    grd1 : $s \in ND \wedge t \in ND$
    grd2 : $s \mapsto t \notin closure(alinks(s))$
    grd3 : $rrq\_pkt \in rrq$
    grd4 : $rrq\_pkt \notin bcast\_rrq$
    grd5 : $source\_rrq(rrq\_pkt) = s$
    grd6 : $target\_rrq(rrq\_pkt) = t$
  THEN
    act1 : $bcast\_rrq := bcast\_rrq \cup \{rrq\_pkt\}$
  END
```

```
EVENT received_rrq
  ANY    t, rrq_pkt, rrp_pkt
  WHERE
    grd1 : $t \in ND$
    grd2 : $rrq\_pkt \in bcast\_rrq$
    grd3 : $target\_rrq(rrq\_pkt) = t$
    grd4 : $source\_rrq(rrq\_pkt) \neq t$
    grd5 : $rrp\_pkt \notin network\_rrp$
  THEN
    act1 : $network\_rrp := network\_rrp \cup$
           $\{rrp\_pkt\}$
    act2 : $bcast\_rrq := bcast\_rrq \setminus \{rrq\_pkt\}$
  END
```

A new event $received\_rrq$ returns a route reply packet rrp_pkt to the initial node with discovered route information from the network. Guards ($grd1-grd4$) of this event state that the broadcasted route request packet (rrq_pkt) is received by the destination node (t) and the source node (s) of the route request packet is not same as the destination node (t). Last guard states that the returning route reply packet (rrp_pkt) is not already received by the route requesting node. Actions of this event state that the destination node returns the route reply packet (rrp_pkt) to the initial node and remove the route request packet (rrq_pkt) from the network.

Note that, together with the events $broadcast\_rrq$, $received\_rrq$ and $update\_rout-ing\_table$ from initial model and all invariants establish **System Requirement 3**.

5.5 Fourth Refinement : Continue Route Discovery Protocol

The route discovery protocol discovers route in the several steps. An address of the original initiator of the request and the target of the request, each route request packet contains a route record, where it is accumulated a record of the sequence of hops taken by the route request packet as it is propagated through the ad hoc network during this route discovery. A new variable ($route\_record\_rrq$) is declared to store the link information at the time of propagation of a route request packet from one node to other node. If the route request receiver node is not the target node then it add the link information to the route record ($route\_record\_rrq$) of the route request packet (rrq_pkt) and again broadcasts it. Similarly, other new variable ($route\_record\_rrp$) used to store the a link information which is collected from the route request packet, when a destination node returns a route reply packet to the initial node. Two more invariants $(inv3, inv4)$ are introduced as safety properties, which represent that the sequence of accumulated node information and the route record information is a subset of all the connected nodes to the source node and a subset of connected links from the source node to all other nodes.

```
$inv1 : route\_record\_rrq \in rrq \rightarrow (ND \leftrightarrow ND)$
$inv2 : route\_record\_rrp \in rrp \rightarrow (ND \leftrightarrow ND)$
$inv3 : \forall rp, al, s \cdot s \in ND \land rp \in rrp \land al \subseteq ND \times ND \land al \in dom(closure) \Rightarrow$
        $dom(route\_record\_rrp(rp)) \subseteq \{x \cdot s \mapsto x \in closure(al) | x\}$
$int4 : \forall al, E, rp \cdot E \subseteq ND \land rp \in rrp \land al \subseteq ND \times ND \land al \in dom(closure) \Rightarrow$
        $route\_record\_rrp(rp) \subseteq closure(E \lhd al)$
```

```
EVENT forward_broadcast
  ANY    x, y, rrq_pkt
  WHERE
    grd1 : $x \in ND \land y \in ND$
    grd2 : $x \mapsto y \in alinks(x) \land$
           $x \mapsto y \in alinks(y)$
    grd3 : $rrq\_pkt \in bcast\_rrq$
    grd4 : $source\_rrq(rrq\_pkt) \neq y$
    grd5 : $target\_rrq(rrq\_pkt) \neq y$
  THEN
    act1 : $route\_record\_rrq(rrq\_pkt) :=$
           $route\_record\_rrq(rrq\_pkt) \cup \{x \mapsto y\}$
    act2 : $bcast\_rrq := bcast\_rrq \cup \{rrq\_pkt\}$
  END
```

```
EVENT update_routing_table
  $\ominus$ $grd4 : E \subseteq \{x | x \in ND \land$
              $s \mapsto x \in closure(ALinks)\}$
  $\ominus$ $grd5 : routeSet \subseteq closure(E \lhd ALinks)$
  $\oplus$ $grd3 : E =$
              $dom(route\_record\_rrp(rrp\_pkt))$
  $\oplus$ $grd4 : routeSet =$
              $route\_record\_rrp(rrp\_pkt)$
EVENT received_rrq
  $\oplus$ $grd6 : s \mapsto t \in$
              $closure(route\_record\_rrq(rrq\_pkt))$
  $\oplus$ $grd7 : route\_record\_rrp(rrp\_pkt) =$
              $route\_record\_rrq(rrq\_pkt)$
```

A new event $forward\_broadcast$ introduces for broadcasting a route request packet to neighboring nodes, when any node is not the destination node for a route discovery process. First two guards state that node x is directly connected with node y and this information is stored by a local routing table of nodes x and y. Next guard ($grd3$) states that a route request packet (rrq_pkt) is already broadcasted and last two guards $(grd4, grd5)$ state that node y is not either source or destination nodes of the route request packet. Two actions of this event, add a new link information $(x \mapsto y)$ as a route record of the route request packet rrq_pkt, and again broadcasts it continue for route discovery process. This process is repeated many times, until the destination node does not receive the route request packet. In this refinement, we introduce some new guards and remove some old guards from events $update\_routing\_table$ and $received\_rrq$.

5.6 Fifth Refinement : Sequence Number

In this last refinement, we introduce a constant $seqNo$ as $seqNo \in rrq \rightarrow \mathbb{N}1$ for representing a sequence number stored in each route request packet. The sequence number is set by the initiator from a locally-maintained sequence number. In order to detect duplicate route requests received packets, each node in the ad hoc network maintains a list of the route request packet that it has recently received on any route request. The route request thus propagates through the ad hoc network until it reaches the destination node, which then replies to the initiator. The original route request packet is received only by those nodes within wireless transmission range of the initiating node, and each of these nodes propagates the request if it is not the target and if the request does not appear to this host to be redundant. Discarding the request as well as recently seen request packet because the address of node is already listed in the route record guarantees that no single copy of the request can propagate around a loop [1]. A new variable $store\_rrq$ is declared as $store\_rrq \in ND \rightarrow \mathbb{P}(rrq)$, which represents a recently seen request table by each nodes. The recently seen request table keeps all visited route request packets information.

A new event $forward\_broadcast\_skip$ is used to discard the route request packet when the request packet is already stored by the recently seen request table ($store\_rrq$). This event is refinement of the event $forward\_broadcast$. Guard ($grd5$) of this event states that route request packet (rrq_pkt) is already received by a node y and it is stored by the recently seen request table ($store\_rrq$). Last guard states that the sequence number ($seqNo$) of the received route request packet is already stored by the recently seen request table ($store\_rrq$) of a node y.

```
EVENT forward_broadcast_skip
  ANY   x, y, rrq_pkt
  WHERE
    grd1 : $x \in ND \wedge y \in ND \wedge x \mapsto y \in alinks(x)$
    grd2 : $rrq\_pkt \in bcast\_rrq$
    grd3 : $source\_rrq(rrq\_pkt) \neq y$
    grd4 : $target\_rrq(rrq\_pkt) \neq y$
    grd5 : $rrq\_pkt \in store\_rrq(y)$
    grd6 : $seqNo(rrq\_pkt) \in$
           $\{p \cdot p \in store\_rrq(y) | seqNo(p)\}$
  THEN
    skip
  END
```

```
EVENT broadcast_rrq
  ⊕ $act2 : store\_rrq(s) := store\_rrq(s) \cup$
          $\{rrq\_pkt\}$
EVENT forward_broadcast
  ⊕ $grd6 : rrq\_pkt \notin store\_rrq(y)$
  ⊕ $grd7 : seqNo(rrq\_pkt) \notin$
          $\{p \cdot p \in store\_rrq(y) | seqNo(p)\}$
  ⊕ $act3 : store\_rrq(y) := store\_rrq(y) \cup$
          $\{rrq\_pkt\}$
```

When route request initiator node (s) broadcasts the route request packet (rrq_pkt), the route request packet is stored in the recently seen request table ($store\_rrq$), which is represented by an extra action in the event $broadcast\_rrq$. Two extra guards ($grd6$, $grd7$) and an action ($act3$) are introduced in the event $forward\_broadcast$. The guards state that a new request packet (rrq_pkt) is received by a node y and sequence number ($seqNo$) of route request packet is different from the recently seen request table ($store\_rrq$). The action ($act3$) states that the intermediate node y stores the route request packet (rrq_pkt) by the recently seen request table ($store\_rrq$).

5.7 Proof Statistics

Table-2 is expressing the proof statistics of the formal development of DSR protocol in the Rodin tool. These statistics measure the size of the model, the proof obligations generated and discharged by the Rodin platform, and those interactively proved. The complete development of the DSR protocol results in 104(100%) proof obligations, in which 83(80%) are proved completely automatically by Rodin tool. The remaining 20(20%) proof obligations are proved interactively by Rodin tool. In the model, many proof obligations are generated in first refinement due to introduction of store and forward architecture for a data packet passing in the dynamic network.

In order to guarantee the correctness of these behaviors, we have established various invariants in stepwise refinement. The stepwise refinement of the DSR protocol helps to achieve a high degree of automatic proof.

Table 2. Proof statistics

Model	Total number of POs	Automatic Proof	Interactive Proof
Abstract Model	16	16(100%)	0(0%)
First Refinement	37	20(55%)	17(45%)
Second Refinement	15	13(91%)	2(9%)
Third Refinement	5	5(100%)	0(0%)
Fourth Refinement	19	17(89%)	2(11%)
Fifth Refinement	12	12(100%)	0(0%)
Total	104	83(80%)	21(20%)

6 Discussion and Conclusion

Discussion: We have found some works on using model checkers and theorem provers to verify properties of routing protocol by O.Wibling et al. [8] and relatively few case studies (e.g., [4,9]) using formal methods to develop different kinds of protocols. Yang et al. [10] have presented both safety and liveness properties of the DSR protocol. The proofs have been mechanically checked using theorem proving tool Isabelle/HOL. Another paper [11] presents a validation model for the DSR protocol using SDL and concludes that *Route Request table* correctly updated after receiving RREP.

This paper contributes to incremental formal development of the DSR protocol using proof-based refinement. The specification is performed in a stepwise manner composing more advanced routing components between the abstract specification and topology. An incremental development helps to verify consistency and correctness of the system. This formal model is designed according to the requirements of the DSR protocol, and provides main characteristics of ad hoc network in form of dynamic networks: nodes can be added and deleted in a dynamic manner. We have introduced several invariants as *safety* properties to verify the system and introduce *liveness* properties that characterize when the system reaches stable states. All these invariants are useful to generate the test cases from formal models, which can be used for testing like route discovery, route updating and response time etcetera.

Conclusion: We have presented a case study for formalizing and reasoning about the DSR protocol in Event-B. Formal development of the DSR protocol is presented in two phases as basic communication protocol and route discovery protocol. In basic communication protocol, we consider the data packets are passing from source node to destination node in changing network. The route discovery protocol is used to find the route from initial node to a destination node. We formalize several different developments, each highlighting different aspects of the problem, making different assumptions and establishing different properties. We consider the case of dynamic environment and express properties for holding the stable states. We have explained our approach for developing DSR protocol using refinement, which allow us to achieve a very high degree of automatic proof. The powerful support is provided by the Rodin tool. Rodin proof is used to generate the proof obligations and to discharge those obligations automatically and interactively. Our approach is the methodology of separation of concerns: first prove the algorithm at an abstract level; then gradually introduce the peculiarity of the specific protocol.

What is important about our approach is that the fundamental properties, we have proved at the beginning, namely the reachability and the uniqueness of a solution, are kept through the refinement process (provided, of course, the required proofs are done). Our different developments reflect not only the many facets of the problem, but also that there was a learning process involved in understanding the problem and its solution. It seems to us that this sort of approach is highly ignored in the literature of protocol developments [10,11] where, most of the time, things are presented in a flat manner directly at the level of the final protocol itself. In addition, the proposed methodology is generic and can be easily applied to other routing protocols for an ad hoc networks. It can also be applied to large-scale system and to extended ad hoc networks like reactive routing protocols.

Acknowledgments. Work of Neeraj Kumar Singh is supported by grant awarded by the Ministry of University and Research.

References

1. Johnson, D.B., Maltz, D.A.: Dynamic Source Routing in Ad Hoc Wireless Networks. In: Mobile Computing. The International Series in Engineering and Computer Science, vol. 353, pp. 153–181. Springer, US (1996) ISSN 0893-3405
2. Project RODIN: Rigorous open development environment for complex systems (2004), `http://rodin-b-sharp.sourceforge.net/` (2004-2007)
3. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
4. Hoang, T.S., Kuruma, H., Basin, D.A., Abrial, J.R.: Developing Topology Discovery in Event-B. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 1–19. Springer, Heidelberg (2009)
5. Cansell, D., Méry, D.: The Event-B Modelling Method: Concepts and Case Studies, pp. 33–140. Springer, Heidelberg (2007); See [12]
6. Leavens, G.T., Abrial, J.R., Batory, D.S., Butler, M.J., Coglio, A., Fisler, K., Hehner, E.C.R., Jones, C.B., Miller, D., Jones, S.L.P., Sitaraman, M., Smith, D.R., Stump, A.: Roadmap for enhanced languages and methods to aid verification. In: Proceedings of the 5th Inter. Conf. on Generative Programming and Component Engineering (GPCE), pp. 221–236 (2006)
7. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
8. Wibling, O., Parrow, J., Pears, A.: Automatized Verification of Ad Hoc Routing Protocols, pp. 343–358. Springer, Heidelberg (2004)
9. Abrial, J.R., Cansell, D., Méry, D.: A Mechanically Proved and Incremental Development of IEEE 1394 Tree Identify Protocol. Formal Asp. Comput. 14(3), 215–227 (2003)
10. Yang, H., Zhang, X., Wang, Y.: A Correctness Proof of the DSR Protocol. In: Cao, J., Stojmenovic, I., Jia, X., Das, S.K. (eds.) MSN 2006. LNCS, vol. 4325, pp. 72–83. Springer, Heidelberg (2006)
11. Cavalli, A., Grepet, C., Maag, S., Tortajada, V.: A Validation Model for the DSR Protocol. In: Proceedings of the 24th ICDCSW 2004, vol. 7, pp. 768–773. IEEE Computer Society, Los Alamitos (2004)
12. Bjørner, D., Henson, M.C. (eds.): Logics of Specification Languages. EATCS Textbook in Computer Science. Springer, Heidelberg (2007)