Int J Softw Tools Technol Transfer (2019) 21:165–181
https://doi.org/10.1007/s10009-017-0463-4

REGULAR PAPER

Formal modeling and analysis of ad hoc Zone Routing Protocol in Event-B

Chunyan Fu[1] · Kougen Zheng[1]

Published online: 7 July 2017

Abstract Ad hoc routing protocols are responsible for searching a route from the source to the destination under the dynamic network topology. Hybrid routing protocols combine the features of proactive and reactive approaches. So, the formal specification of a hybrid routing protocol in the dynamic network environment is a challenge. In this paper, we formally analyze the Zone Routing Protocol (ZRP), a hybrid routing framework, using Event-B. We develop the formal specification by the refinement mechanism. It allows us to gradually model the network environment, the construction of routing zones, route discovery based on bordercasting service and routing update. We prove the stabilization property in the inactive environment. In addition, we demonstrate that discovered routes hold the loop freedom and validity in each reachable system state. To present that the formalization is consistent with the informally expressed requirements, we adopt an animator, ProB, to validate our model. Our work provides reference to analyze extensions of the ZRP and other hybrid routing protocols.

Keywords Formal verification · Hybrid routing protocols · Zone Routing Protocol · Event-B · Refinement

1 Introduction

An ad hoc network is built on a number of mobile nodes which work in cooperation without any fixed infrastructure. Each node is considered as both a host and a router. Since nodes can move around freely, the network topology may change at any time. Therefore, ad hoc routing protocols have to find a route from the source to the destination under the dynamic environment.

In terms of the routing information update mechanisms, the existing routing protocols can be classified into three categories [1]: proactive routing protocols, reactive routing protocols and hybrid routing protocols. Proactive (table driven) ones maintain a fresh view of the topological map via periodic and/or event-triggered routing updates, e.g., the Destination Sequenced Distance Vector (DSDV) [21] protocol and the Optimized Link State Routing (OLSRv2) [10] protocol, while reactive (on demand) protocols initiate a route discovery on demand, such as the Dynamic Source Routing (DSR) [16] protocol and the Ad-hoc On-demand Distance Vector Routing (AODV) [22] protocol. Hybrid routing protocols combine the features of above two approaches, such as the Zone Routing Protocol (ZRP) [14]. A major issue is how to guarantee their desirable goal, discovering the required routes, in the active environment.

Formal methods, especially formal verification, can enhance the quality of a verified system. Formal verification techniques can determine whether a system has or has not a given property by strict mathematical proof. So far, there has been much work on using model checkers or theorem provers to verify routing protocols [6,7,19,24,25]. To the best of our knowledge, few people focus on the formal verification for hybrid routing protocols. In this paper, we present a formal specification for the ZRP in Event-B and prove the correctness of the route discovery mechanism. Event-B [3], a refinement-based method, allows system details to be gradually added to the corresponding models. Any properties that are already proved to hold in the early models are ensured to hold in the later models. It relieves the user of the

✉ Chunyan Fu
fucy@zju.edu.cn

Kougen Zheng
zkg@cs.zju.edu.cn

[1] College of Computer Science and Technology, Zhejiang University, Hangzhou, Zhejiang, China

burden of modeling and proving. With a friendly assistant Rodin [4] tool, the proof task is automatically divided into manageable pieces that depend on the structure of a model. Moreover, this approach has been applied to systems from various domains [3,5].

With the ZRP, each node regularly broadcasts its link state information throughout its routing zone. To model the periodic broadcast behavior, it requires to introduce some time constraints in the formalization. Additionally, the formalization has to limit the propagation scope for a node's link state information within its routing zone. A node processes route requests with bordercasting service, rather than broadcasting usually adopted by reactive protocols. Furthermore, a node updates its routing table if there exists changes in its routing zone or it receives route replies carrying discovered routes.

Consequently, the formalization for the ZRP is more complicated than purely a proactive or a reactive protocol. This is a significant issue that needs to be addressed and an interesting challenge as to how to formally specify it.

We analyze the system via refinement. It permits us to develop a system from abstract to concrete. The correctness for refinements is assured through discharging some proof obligations. Here, we concentrate on modeling the highlighted aspects of the system: the connectivity of the network topology; the construction of routing zones according to the periodically exchanged neighbor information; the route discovery with the bordercasting service; the routing update in the reactive component. We also prove the stabilization property in the stable environment and the preservation of system properties, e.g., the loop freedom and validity of discovered routes, in each reachable system state. For the purpose of ensuring that our model has formalized the system requirements, we utilize the ProB, an animation tool, which is available for the Rodin, to validate our model.

The paper is organized as follows. The introduction about the Event-B method is presented in sect. 2. In sect. 3, we give an overview of the ZRP framework. Furthermore, we present system requirements and environment assumptions considered in our development. We show the entire formal development process, including the formalization with Event-B and the validation with the ProB, in sect. 4. At last, we discuss related work and draw a conclusion in sect. 5.

2 Modeling in Event-B

2.1 Event-B Method

Event-B is a state-of-the-art formal method for system level modeling and analysis [3]. It is a simplification and extension of the B-Method [2]. Event-B is developed on basis of set theory and first-order predicate logic. The syntax specification of the Event-B language is presented in [20]. By now, this approach has been applied to verify many complex systems [3,5,9].

An Event-B model is composed of two structures: contexts and machines. Contexts describe the static part of a model, and machines specify the dynamic part of a model. The explicit definitions for context and machine are as follows.

Definition 1 A context structure is a tuple $(\mathcal{S}, \mathcal{C}, \mathcal{A}, \mathcal{T})$, where $\mathcal{S}$ is a set of user-defined sets, $\mathcal{C}$ is a set of constants, $\mathcal{A}$ is a set of axioms which $\mathcal{S}$ and $\mathcal{C}$ obey, $\mathcal{T}$ is a set of theorems.

Definition 2 A machine structure is a tuple $(\mathcal{V}, \mathcal{I}, \mathcal{T}, \mathcal{VA}, \mathcal{E})$, where

- $\mathcal{V}$ is a set of variables which describe the states of a system,
- $\mathcal{I}$ is a set of predicates which declare the properties of variables. The predicates in $\mathcal{I}$ should be preserved in each reachable system state,
- $\mathcal{T}$ is a set of theorems which have to be proved in the machine,
- $\mathcal{VA}$ is a set of variants defined in the machine containing some convergent events,
- $\mathcal{E}$ is a set of events which model the behaviors of a system.

An event, a relation on the state set or a before-after predicate, has two main parts: guards which state preconditions for event execution and actions which assign values to variables. The most general form of an event is as follows:

$$\textbf{any } x \textbf{ where } G(x, v) \textbf{ then } A(x, v) \textbf{ end} \qquad (1)$$

where x represents a collection of parameters, v represents a set of variables, $G(x, v)$ represents for the conjunction of some guards and $A(x, v)$ represents for the actions. The parameters and guards are optional. So, an event may be in a simplified form without parameters or guards. Actions contain several assignments that are supposed to happen simultaneously. Each assignment is a before-after predicate describing the change of a variable.

The Rodin [4] platform provides a support for the construction and verification of Event-B models. The verification is performed by proving (automatically or interactively) the automatically generated proof obligations. Additionally, this platform is extensible and configurable [8] since it has a plugin architecture. One plugin provided is ProB [17], a tool for model animation and model checking.

2.2 Model refinement

Refinement, a powerful modeling mechanism, allows us to build a model in a step-by-step manner [3]. Through the refinement, both context and machine can be extended to develop the concrete models. A context can extend one or more contexts. Then, it contains the static information defined in extended ones. A machine can refine only one existing machine, while it can see several contexts. For the refined model, we can replace abstract variables by concrete ones and refine abstract events. For an existing event, we can preserve the event, split it into several concrete events or refine it to another event by strengthening guards or adding new actions. Moreover, we can also introduce new events which refine the skip event doing nothing.

To ensure the correctness of refinements, we need to prove some proof obligations to guarantee that (1) the abstract event is enabled when the concrete one is enabled, referred to as guard strengthening, (2) the gluing invariants relating to the variables of the concrete machine and the abstract machine are preserved, (3) each action in the abstract event is stimulated in the corresponding concrete event. We show the generated rules for the proof obligations in the following definition.

Definition 3 Let $\mathcal{M}1$, $\mathcal{M}2$ be machines. Let e_1, e_2 be events such that $e_1 \in \mathcal{M}1$, $e_2 \in \mathcal{M}2$. If $\mathcal{M}2$ refines $\mathcal{M}1$, in addition, e_2 refines e_1 and

- $\mathcal{A}$ is the set of all axioms seen by $\mathcal{M}2$,
- $\mathcal{T}$ is the set of theorems seen by $\mathcal{M}2$ or defined in $\mathcal{M}1$, $\mathcal{M}2$,
- $\mathcal{I}$ is the set of invariants introduced in $\mathcal{M}1$, $\mathcal{M}2$,
- $\mathcal{G}$ is the set of guards of e_2,
- $\mathcal{J}$ is a gluing invariant introduced in $\mathcal{M}2$ and e_2 modifies $\mathcal{J}$,
- $\mathcal{BA}_1$, an action of e_1, is the before-after predicate for the assignment of an abstract variable, the corresponding concrete behavior is given by $\mathcal{BA}_2$, an action of e_2.

Then, we have to prove

- $\mathcal{A}, \mathcal{T}, \mathcal{I}, \mathcal{G} \vdash grds$, where $grds$ is the conjunction of guards of e_1;
- $\mathcal{A}, \mathcal{T}, \mathcal{I}, \mathcal{G} \vdash \mathcal{J}'$, where $\mathcal{J}'$ is the modified $\mathcal{J}$;
- $\mathcal{A}, \mathcal{T}, \mathcal{I}, \mathcal{G}, \mathcal{BA}_2 \vdash \mathcal{BA}_1$.

3 Overview of the ZRP

3.1 Informal description

The hybrid ZRP protocol combines some features of the proactive and reactive schemes [14]. Note that it is only a framework instead of a specific routing protocol. It includes two components: the Intrazone Routing Protocol (IARP) [13] used within routing zones and the Interzone Routing Protocol (IERP) [12] used beyond routing zones. Both of them are not specific protocols, they are families of proactive link state routing protocols and reactive routing protocols, respectively. With a proper zone radius, its performance is at least as well as the pure constituents.

By the IARP, each node maintains the local routing information on the basis of the periodically exchanged neighbor discovery messages. A node's routing zone is a set of nodes whose minimum distance (in hops) from this node is no greater than a value called zone radius. For building routing zones, each node has to get the knowledge of its neighbors. Here, the current neighbor information is provided by a separate Neighbor Discovery Protocol (NDP). Then, the node sends its own link state throughout the limited scope restricted by the zone radius.

The IERP, the reactive part of ZRP, is responsible for the route discovery and route maintenance [12]. With the network topology provided by the IARP, the routes to the destinations within a routing zone are available. If a destination is outside the routing zone, the source with IERP initiates the route discovery based on bordercasting service known as the Bordercast Resolution Protocol (BRP) [11].

The route discovery process consists of the route request phase and the route reply phase. A source node searches routes by bordercasting. It constructs its bordercast tree which is a multicast tree spanning all the *peripheral nodes* whose minimal distance from the source is exactly equal to the zone radius. Then, it forwards the route request to the neighbors in the bordercast tree. When the request has been forwarded, the routing zone of source is covered. A node in the bordercast tree of the previous forwarding node is referred to as an *intended recipient*. Note that only intended recipients are required to process the request. If the intended receiver has a valid route to the destination within its routing zone, it sends a route reply containing the complete route toward the source node. Otherwise, it builds a bordercast tree which spans all uncovered peripheral nodes around it and bordercasts the route request. By this way, it prunes the branches leading to the peripheral nodes inside covered regions (routing zones of previous bordercasting nodes). After forwarding this request, it marks the routing zone as covered. With the routing zones and the bordercasting service, it is efficient to probe the entire network topology.

After a route has been discovered, knowledge of the routing zone can be utilized to bypass link failures and identify suboptimal route segments [12].

3.2 System requirements

Rather than modifying a proactive link state protocol as the IARP and a reactive protocol as the IERP, we analyze the system based on the description in [11–13]. In our development, we adopt the sequence number, tracking the link state history of a node described in [13], to avoid dealing with old link state information. The loop freedom of discovered routes is ensured by the reactive protocol, such as DSR. In addition, we consider a uniform zone radius for the whole network. If the zone radius is one hop, ZRP turns into a reactive protocol. To focus on the special aspects of the ZRP, we pay our attention on the general situation that the zone radius is greater than one hop.

The main system requirements are:

REQ-1. The Zone Routing Protocol is a hybrid ad hoc network routing framework. Its goal is to discover new routes on demand with the knowledge of local zone in the dynamic network environment.
REQ-2. The route between two distinct nodes can be eventually discovered if there exists a valid route in the real network environment.

The **REQ-1** describes the overall purpose of the protocol. Since the network topology may change frequently, a connected link can become down after a few seconds. Hence, the discovered route may be invalid when the query source receives it. But the route between the source and the destination can be eventually discovered if there exists a path between them in the network topology. This is the correctness requirement about route discovery stated by [24]. **REQ-2** specifies this requirement. In what follows, we present the definition of correct route considered in this paper.

Definition 4 (Correct Route) Let $v_1, \ldots, v_k$, with k a positive natural number and $k > 1$, be arbitrary distinct nodes in the network. The discovered route denoting by $route = \{v_1 \mapsto v_2, \ldots, v_{k-1} \mapsto v_k\}$ for the source s and destination t is correct if it satisfies

1. s and t are nodes in the network;
2. for all $v_i \mapsto v_{i+1} \in route$ with $i \in 1..k-1$, v_i, v_{i+1} are nodes belonging to the network;
3. there is no loop in the $route$;
4. $route$ is a complete path from s to t.

REQ-3. Each node can periodically obtain the link status with its neighbors and then exchange the neighbor information with other nodes.
REQ-4. The scope of link state update is limited by the routing zones.
REQ-5. Each node maintains the up-to-date topology map within its routing zone. So, it has routes to the destinations within its routing zone.

A node periodically consults its neighbor information provided by the NDP. Then, it broadcasts its link state throughout its routing zone. Based on **REQ-3** to **REQ-5**, each node has a fresh view of its local zone. Although the view of nodes may be inconsistent with the actual network topology, those requirements ensure the view of nodes is more closer to the real environment.

REQ-6. Route discovery relies on the routing zones. The source node initiates the route discovery if there is no route to the destination in its routing table.

By the bordercasting service, the source node forwards the route request to a subset of its neighbors. That is not all neighbors can receive this request.

REQ-7. The route request can only be received by the intended recipients determined by the previous forwarding nodes.
REQ-8. The reaction of an intended recipient depends on whether the destination is in its routing zone and can be one of the following: (1) upon within the routing zone, it delivers a route reply to the source, (2) otherwise, it appends its address to the accumulated path and bordercasts the request.
REQ-9. With the zone-based query control, a route request can be guided away from the query source and the regions covered by the request.

According to the zone-based query control, a node shall mark its routing zone as covered if it has forwarded the request or the reply. Then, it is unnecessary to process the request again.

The next requirement concerns the routing update.

REQ-10. Every node updates its routing information if there exists changes within its routing zone or it relays a route reply packet.

When the source node receives a valid route reply, it implies that a route has been found in the dynamically changing environment.

3.3 Environment assumptions

Before we engage in the formal development, we list the environment assumptions as follows:

ENV-1.There are finitely many nodes in the network.
ENV-2.The bidirectional links between some pairs of different nodes may be up or down. There is no intermediate status.
ENV-3. When a link from node m to node n becomes up or broken, m is aware of the changes.
ENV-4. If the environment is inactive for a long time, then each node has a correct view of its surrounding topology map.

We assume the links are bidirectional in the network (**ENV-2**). That is if node m and n are connected, then m can send packets to n and vice versa. Thus, it is able to propagate a route reply toward the source with the discovered route. In what follows, we refer to a link from m to n as an outward link from m. If m and n directly connect, then it indicates that directed links $m \mapsto n$ and $n \mapsto m$ are up in the network environment. Based on the **ENV-3**, each node can sense its outward links. Note that it does not require that a node immediately detects the changes. The **ENV-4** describes the system will eventually reach the temporary stable state if the network environment is static for a long time. Then, each node's view about its routing zone is consistent with the actual network topology.

4 Formal development

In this section, we present the formal development of the ZRP. Firstly, we present the refinement strategy for our analysis. Through the step-by-step refinement, we derive a model formalizing the goal of the protocol (**REQ-1**) and the correctness requirement **REQ-2**. At last, we show the validation of our model.

Initial Model. This model constructs the dynamic network architecture.

Refinement 1. In this step, we introduce the abstract update events for the link state table and routing table, respectively. Moreover, we consider the stable state of the system under the quiescent environment.

Refinement 2. We formalize the updating for link state table in detail. This refinement briefly introduces that each node periodically broadcasts its link state. The refresh event excludes links whose survival time exceeds the longest time to live in the link state table.

Refinement 3. With a uniform zone radius, we model the construction of routing zones based on Refinement 2. To avoid processing the old link information, we utilize the sequence number to keep track of each packet.

Refinement 4. To take into account the distributed behavior of each node, we use variables to record the being transmitted information on the connected links.

Refinement 5. In this refinement, we analyze the route request phase briefly. The bordercast delivery process without query control is formalized.

Refinement 6. We develop concrete events to state that an intended recipient deals with the received route request.

Refinement 7. In this refinement, we consider the zone-based query control during the route request phase.

Refinement 8. The route reply phase is modeled in this refinement. A node updates its routing table based on the notification derived from IARP or a received route reply. We refine the routing table update event in this step.

The Event-B formalism allows us to model the interaction between the system and its environment. So, we formalize the dynamic network environment in the initial model. Figure 1 shows that each subsequent refinement considers different requirements. For instance, the refinement 2 ensures the **ENV-3** and **REQ-3**, and the refinement 5 considers the requirements **REQ-6** and **REQ-7**. Based on the Definition 4, we prove points 1 and 2 in the refinement 1, the loop freedom and validity of discovered routes in the refinement 5 and refinement 8, respectively. So, the correctness of discovered routes is ensured in the final model. The **REQ-1** is fairly general, and in fact taken account at each refinement step. As a result, the final refinement completes the requirements **REQ-1** and **REQ-2**.

Since the space is limited, we just present some parts of the development.

4.1 Modeling the environment

In the initial context, we introduce a carrier set *Nodes* denoting the set of all nodes in the network. According to **ENV-1**,

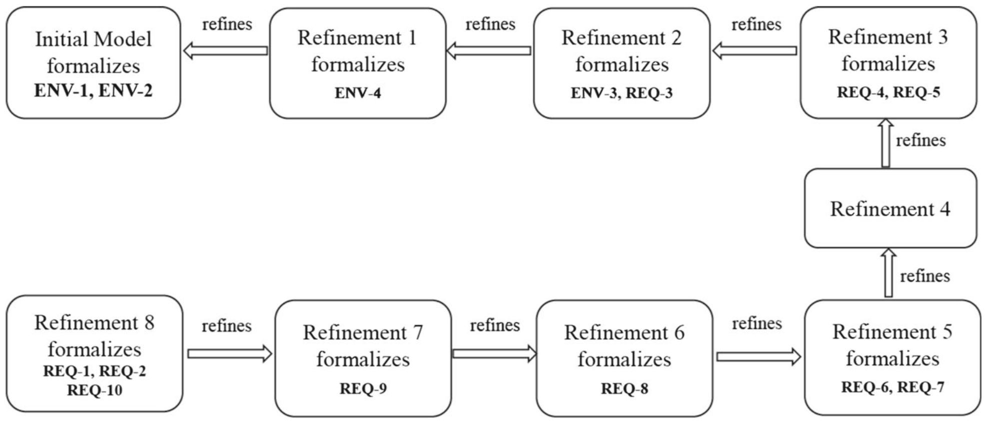
Initial Model formalizes ENV-1, ENV-2
refines
Refinement 1 formalizes ENV-4
refines
Refinement 2 formalizes ENV-3, REQ-3
refines
Refinement 3 formalizes REQ-4, REQ-5
refines
Refinement 4
refines
Refinement 8 formalizes REQ-1, REQ-2 REQ-10
refines
Refinement 7 formalizes REQ-9
refines
Refinement 6 formalizes REQ-8
refines
Refinement 5 formalizes REQ-6, REQ-7

Fig. 1 Overview of the formal development

we axiomatize *Nodes* as a finite set. Then, we can obtain the following theorem: a relation defined over finite sets is finite ($thm1$). It shall facilitate the following proof task, e.g., the well-definedness proof for parameter $collection$ of receiveRequest_HasRoute presented in the **Sixth Refinement**.

```
$axm1 : \text{finite}(Nodes)$
$thm1 : \forall a, b, f \cdot a \in \mathbb{P}(Nodes) \land b \in \mathbb{P}(Nodes) \land f \in a \leftrightarrow b$
        $\Rightarrow (\text{finite}(a) \land \text{finite}(b) \Rightarrow \text{finite}(f))$
```

To characterize the connectivity of the dynamic network topology, we use the variable $NeighborLink$ to record the currently up link information in the network. The variable $DNeighborLink$ denotes the set of currently down links that were up. From $thm1$ and $NeighborLink$, we can see that there are finitely many connected links in the network topology. Hence, it is possible to propagate a route request throughout the whole network. In order to formalize the changes about the network topology, we construct three events: Add_NewLink, Add_BrokenLink and Remove_Link. The first two state that two distinct nodes become connect, and then the corresponding links are added to $NeighborLink$. Remove_Link removes the invalid links, up links at some point in the past, from $NeighborLink$. We present the events as follows. Guard $node1 \neq node2$ ensures that a pair of nodes are distinct. The fourth guard in Add_NewLink states that the current network topology does not contain the up links between $node1$ and $node2$. Add_BrokenLink also requires this condition. The difference between these two events is that the former considers new links, not initially broken, and the latter considers some up links, broken in the past.

```
Add_NewLink
 any $node1$ $node2$ where
   $node1 \in Nodes$
   $node2 \in Nodes$
   $node1 \neq node2$
   $node1 \mapsto node2 \notin NeighborLink$
   $\land node2 \mapsto node1 \notin NeighborLink$
   $node1 \mapsto node2 \notin DNeighborLink$
   $\land node2 \mapsto node1 \notin DNeighborLink$
 then
   $NeighborLink := NeighborLink \cup$
                $\{node1 \mapsto node2, node2 \mapsto node1\}$
```

```
Add_BrokenLink
 any $node1$ $node2$ where
   $node1 \in Nodes$
   $node2 \in Nodes$
   $node1 \neq node2$
   $node1 \mapsto node2 \notin NeighborLink$
   $\land node2 \mapsto node1 \notin NeighborLink$
   $node1 \mapsto node2 \in DNeighborLink$
   $\land node2 \mapsto node1 \in DNeighborLink$
 then
   $NeighborLink := NeighborLink \cup$
                $\{node1 \mapsto node2, node2 \mapsto node1\}$
   $DNeighborLink := DNeighborLink \setminus$
                $\{node1 \mapsto node2, node2 \mapsto node1\}$
```

Remove_Link demands that the removed links are in $NeighborLink$. Then, it adds these links to the set $DNeighborLink$.

```
Remove_Link
 any $node1$ $node2$ where
   $node1 \in Nodes$
   $node2 \in Nodes$
   $node1 \neq node2$
   $node1 \mapsto node2 \in NeighborLink$
   $\land node2 \mapsto node1 \in NeighborLink$
 then
   $NeighborLink := NeighborLink \setminus$
                $\{node1 \mapsto node2, node2 \mapsto node1\}$
   $DNeighborLink := DNeighborLink \cup$
                $\{node1 \mapsto node2, node2 \mapsto node1\}$
```

To constitute the **ENV-2**, we also introduce some invariants. Each link in $NeighborLink$ consists of distinct nodes ($inv3$). Additionally, if $NeighborLink$ contains a directed link $node1 \mapsto node2$, then the reverse of this link, i.e., $node2 \mapsto node1$, is also in $NeighborLink$ ($inv4$). $NeighborLink$ and $DNeighborLink$ are disjoint ($inv5$).

```
$inv1 : NeighborLink \in Nodes \leftrightarrow Nodes$
$inv2 : DNeighborLink \in Nodes \leftrightarrow Nodes$
$inv3 : \forall node1, node2 \cdot node1 \in Nodes \land node2 \in Nodes \land$
    $node1 \mapsto node2 \in NeighborLink \Rightarrow node1 \neq node2$
$inv4 : \forall node1, node2 \cdot node1 \in Nodes \land node2 \in Nodes \land$
      $node1 \mapsto node2 \in NeighborLink$
      $\Rightarrow node2 \mapsto node1 \in NeighborLink$
$inv5 : NeighborLink \cap DNeighborLink = \emptyset$
```

4.2 Construction of routing zones

The main aim of our development is to construct a model that meets the system requirements and environment assumptions. We abstractly formalize the final result of the protocol in the first refinement. Then, following refinements model the construction of routing zones.

First Refinement. We define *closure* to represent the transitive closure of links. Its properties are shown as follows.

```
$axm1 : closure \in (Nodes \leftrightarrow Nodes) \rightarrow (Nodes \leftrightarrow Nodes)$
$axm2 : \forall r \cdot r \subseteq closure(r)$
$axm3 : \forall r \cdot closure(r); r \subseteq closure(r)$
$axm4 : \forall r, s \cdot r \subseteq s \land s; r \subseteq s \Rightarrow closure(r) \subseteq s$
$axm5 : \forall r \cdot closure(r); closure(r) \subseteq closure(r)$
```

We define a variable $LinkStateTable$ to specify the link state information recorded in each node. A node updates this table when it periodically consults its neighbor information, receives another node's neighbor information or removes some expired links. With the purpose of updating a node's link state table, the abstract updateLinks requires to remove some useless links and add some new links. In terms of above mentioned cases, the $addlinks$ and $removelinks$ are two disjoint sets except in the second one. To model this case, we choose to remove the stored old neighbor information and then add the received neighbor information. So, we decide to first remove some links and then add useful links. We refine this event in the next step.

```
updateLinks
 any $node\ addlinks\ removelinks$ where
   $node \in Nodes$
   $addlinks \in Nodes \leftrightarrow Nodes$
   $removelinks \in Nodes \leftrightarrow Nodes$
 then
   $LinkStateTable(node) := (LinkStateTable(node)$
                  $\setminus removelinks) \cup addlinks$
```

We formalize a node's routing table as a collection of links. The variable $RoutingTable$ specifies this information.

```
$inv1 : LinkStateTable \in Nodes \rightarrow (Nodes \leftrightarrow Nodes)$
$inv2 : RoutingTable \in Nodes \rightarrow (Nodes \leftrightarrow Nodes)$
$inv3 : \forall n \cdot n \in Nodes \Rightarrow RoutingTable(n) \subseteq$
      $Nodes \times Nodes$
```

The updateRoutingTable event models the routing update for $node$. If there is no links to add or remove, the routing table of $node$ is unchanged. Each node updates its routing table if it receives a route reply with a discovered route or detects some changes within its routing zone. In the latter case, the $addRoutes$ represents the set of links in the current routing zone, and the $removeRoutes$ denotes the set of links in the original routing zone. There may be some links in the intersection of $addRoutes$ and $removeRoutes$. So, we prefer to first remove those old routes and then add current routes. This abstract event just states the final result of the system. From the guards, we can prove $inv3$. We gradually introduce the details in the subsequent refinements.

```
updateRoutingTable
 any $node\ addRoutes\ removeRoutes$
 where
   $node \in Nodes$
   $addRoutes \in Nodes \leftrightarrow Nodes$
   $removeRoutes \in Nodes \leftrightarrow Nodes$
   $\neg(addRoutes = \emptyset \wedge removeRoutes = \emptyset)$
 then
   $RoutingTable(node) := (RoutingTable(node)$
                  $\setminus removeRoutes) \cup addRoutes$
```

Under the inactive network environment, the system will eventually reach a stable state. It means that the link state information stored in each node is consistent with the physical environment. The notion of system stability is an instance of the notion of a stable system property P [18], which satisfies if P is true of any reachable state s of a system, then P is true of all system states reachable from s. In terms of the so-called observer event [15], we introduce the stabilize event, with the aim of defining the notion of a stable system state. It has no effect on the system state as it does nothing. The parameter $nodes$ can indicate whether a node is in the propagation scope of a broadcasting node. The second guard shows that each node has the correct view about its link state. The third guard declares that if there is a route from m to n and n is in the propagation scope of m, then n has the correct view of m's link state. This event is abstract as it does not give a precise definition for the parameter $nodes$. We can state that the system is in a stable state if the stabilize event is enabled. This step formalizes **ENV-4**.

```
stabilize
 any $nodes$ where
   $nodes \in Nodes \rightarrow \mathbb{P}(Nodes)$
   $\forall m, n \cdot m \mapsto n \in NeighborLink$
        $\Leftrightarrow m \mapsto n \in LinkStateTable(m)$
   $\forall m, n \cdot m \mapsto n \in closure(NeighborLink) \wedge n \in nodes(m)$
        $\Rightarrow (\forall x \cdot m \mapsto x \in LinkStateTable(n)$
        $\Leftrightarrow m \mapsto x \in LinkStateTable(m))$
```

Second Refinement. With the IARP, every node exchanges its neighbor information with other nodes and keeps a fresh view of its routing zone. In this refinement, we shall model the exchange procedure without the limitation in scope.

We define some constants to describe the static part of the model. $lifetime$ represents the lifetime for links in the link state table. $period$ is the broadcast period, satisfying $lifetime > period$. For eliminating outdated links, we introduce a variable $InsertionTime$ to keep track of the insertion time for each link in $LinkStateTable$. The $Time$, a positive natural number, denotes the current time in the model. For $n \in Nodes$, if link $l \in LinkStateTable(n)$, then its corresponding insertion time is stored in $InsertionTime(n)$ and vice versa ($inv4$). dom(r) is the set of all elements in the domain of a relation r. Moreover, the broadcast time of each node, recorded by $BroadcastTime$, is strictly not greater than $Time$ ($inv5$). We state the relationship between the network environment and the link state information with the following invariants. A neighbor link of a node is currently up or up at some point ($inv6$). In the dynamic environment, namely some nodes do not immediately update its link state information when there exists changes in the environment, the node's view of its neighbors does not coincide with the network environment ($inv7$, $inv8$).

```
$inv1 : InsertionTime \in Nodes \rightarrow ((Nodes \times Nodes) \nrightarrow \mathbb{N})$
$inv2 : Time \in \mathbb{N}_1$
$inv3 : BroadcastTime \in Nodes \rightarrow \mathbb{N}_1$
$inv4 : \forall n \cdot n \in Nodes \Rightarrow (\forall l \cdot l \in LinkStateTable(n)$
      $\Leftrightarrow l \in \mathrm{dom}(InsertionTime(n)))$
$inv5 : \forall n \cdot n \in Nodes \Rightarrow BroadcastTime(n) \leq Time$
$inv6 : \forall m, n \cdot m \mapsto n \in LinkStateTable(m) \Rightarrow (m \mapsto n$
      $\in NeighborLink \vee m \mapsto n \in DNeighborLink)$
$inv7 : \exists m, n \cdot m \mapsto n \in LinkStateTable(m) \wedge m \mapsto n \notin$
      $NeighborLink \wedge m \mapsto n \in DNeighborLink$
      $\Rightarrow \neg(\forall q, p \cdot q \mapsto p \in NeighborLink \Leftrightarrow q \mapsto p$
      $\in LinkStateTable(q))$
$inv8 : \exists m, n \cdot m \mapsto n \notin LinkStateTable(m) \wedge m \mapsto n \in$
      $NeighborLink \Rightarrow \neg(\forall q, p \cdot q \mapsto p \in NeighborLink$
      $\Leftrightarrow q \mapsto p \in LinkStateTable(q))$
```

The abstract updateLinks is split into: obtainLinks, transferLinks and refreshLinks. Parameters $addlinks$ and $removelinks$ should be precisely expressed in each concrete event. Event obtainLinks states that a node consults its neighbor link information.[1] Each node can be aware of the link status with its neighbors. In this event, the $addlinks$ is the set of links between $node$ and its new neighbors. Set $removelinks$ collects links between $node$ and its invalid neighbors. Since it is unnecessary to manage the insertion

[1] $\oplus$ indicates the added guard or action, $\ominus$ indicates the removed guard or action.

time of $removelinks$, we utilize $rest$ to keep the insertion time for residual links in $node$'s link state table. Stimulated by the broadcast period, $node$ also updates the insertion time for neighbor links and the broadcasting time. The $\mathbin{\lhd\!\!-}$ denotes relational overwrite.

```
obtainLinks refines updateLinks
 any $node\ addlinks\ removelinks\ rest$
 where
  $\oplus addlinks = \{p \mapsto m | p = node \wedge p \mapsto m \in NeighborLink$
   $\wedge p \mapsto m \notin LinkStateTable(p)\}$
  $\oplus removelinks = \{p \mapsto m | p = node \wedge p \mapsto m \notin NeighborLink$
   $\wedge p \mapsto m \in LinkStateTable(p)\}$
  $\oplus Time - BroadcastTime(node) \geq period$
  $\oplus rest = removelinks \mathbin{\ntriangleleft} InsertionTime(node)$
 then
  $\oplus InsertionTime(node) := rest \mathbin{\lhd\!\!-} (((\{node\} \lhd \mathrm{dom}(rest)) \cup$
        $addlinks) \times \{Time\})$
  $\oplus BroadcastTime(node) := Time$
```

Note that the routing zone notion is not considered here. transferLinks models the situation that a node ($sender$) transfers a node's link state to one of its neighbors ($receiver$). By this event, each node exchanges its link state with others. Before it records the insertion time for the received links, recording by $addlinks$, it removes the old link information of $transferNode$, recording by $removelinks$.

```
transferLinks refines updateLinks
 any $sender\ receiver\ transferNode\ receivedlinks$
   $addlinks\ removelinks$
 where
  $\ominus node \in Nodes$
  $\oplus sender \in Nodes \wedge receiver \in Nodes \wedge$
   $transferNode \in Nodes$
  $\oplus sender \neq receiver \wedge receiver \neq transferNode$
  $\oplus sender \mapsto receiver \in LinkStateTable(sender)$
  $\oplus receivedlinks \in Nodes \leftrightarrow Nodes \wedge$
    $\mathrm{dom}(receivedlinks) = \{transferNode\}$
  $\oplus addlinks = receivedlinks$
  $\oplus removelinks = \{transferNode\} \lhd$
        $LinkStateTable(receiver)$
 with
  $node = receiver$
 then
  $\ominus LinkStateTable(node) := (LinkStateTable(node) \setminus$
        $removelinks) \cup addlinks$
  $\oplus LinkStateTable(receiver) := (LinkStateTable(receiver) \setminus$
        $removelinks) \cup addlinks$
  $\oplus InsertionTime(receiver) := (removelinks \mathbin{\ntriangleleft}$
        $InsertionTime(receiver)) \cup (addlinks \times \{Time\})$
```

```
refreshLinks refines updateLinks
 any $node\ oldlinks$
 where
  $\oplus oldlinks = \{x \mapsto y | x \mapsto y \in LinkStateTable(node) \wedge$
  $x \neq node \wedge$
  $(Time - InsertionTime(node)(x \mapsto y) \geq lifetime)\}$
 with
  $addlinks = \emptyset$
  $removelinks = oldlinks$
 then
  $\ominus LinkStateTable(node) := (LinkStateTable(node)$
        $\setminus removelinks) \cup addlinks$
  $\oplus LinkStateTable(node) := LinkStateTable(node) \setminus oldlinks$
  $\oplus InsertionTime(node) := oldlinks \mathbin{\ntriangleleft} InsertionTime(node)$
```

The refreshLinks event updates the link state table by excluding expired links whose link sources have moved out of the node's routing zone, $removelinks$ noting down this knowledge. $addlinks$ is an empty set in this case. The time growth process is modeled by a new event timeClock with an increment $tick \in \mathbb{N}_1$. The **REQ-3** and **ENV-3** are implemented in this step.

Third Refinement. In ZRP, every node maintains the routing information within its routing zone rather than the whole network. In this refinement, we focus on the construction of routing zones.

The constant $zoneRadius$ denotes the zone radius and $zoneRadius > 1$. The scope of a link state update is limited by the TTL (time to live) [13] carried in the link state packets. We define a variable TTL to record the TTL value along with packets transmission. When a node advertises its neighbor information, it initializes the TTL value as $zoneRadius - 1$. If the packet is received by a node, the value is decremented. The broadcasting shall terminate until the value is equal to 0. This protocol utilizes the sequence number to track the history of each link state packet. So, we define the variable $SeqNum$. $inv3$ states that the recorded sequence number for each received link state packet is greater than zero. A node n, within the routing zone of an arbitrary node m, having a distinct neighbor view of m implies that n does not receive the current neighbor information of m ($inv4$).

```
$inv1 : TTL \in Nodes \rightarrow (Nodes \rightarrow \mathbb{N})$
$inv2 : SeqNum \in Nodes \rightarrow (Nodes \rightarrow \mathbb{N})$
$inv3 : \forall m, p \cdot p \in \mathrm{dom}(LinkStateTable(m))$
     $\Rightarrow SeqNum(m)(p) > 0$
$inv4 : \forall m, n \cdot n \in \mathrm{dom}(LinkStateTable(m)) \wedge m \mapsto n \in$
   $closure(LinkStateTable(m)) \wedge (\exists x \cdot m \mapsto x \in$
   $LinkStateTable(m) \wedge \neg m \mapsto x \in LinkStateTable(n))$
   $\Rightarrow \neg(\forall x \cdot m \mapsto x \in LinkStateTable(m) \Leftrightarrow m \mapsto x \in$
   $LinkStateTable(n))$
```

We refine the obtainLinks to initialize the TTL value and increase the sequence number. If a node receives a node's link state packet with a smaller sequence number, then it discards this information. The discardLinks event specifies this behavior. Otherwise, the node processes this packet by transferLinks which decreases the TTL value and notes down the sequence number. In addition, the receiver just deals with the received information with the TTL greater than 0.

```
transferLinks refines transferLinks
 any $sender\ receiver\ transferNode\ receivedlinks$
   $addlinks\ removelinks\ receivedTTL\ receivedSeqNum$
 where
  $\oplus receivedTTL > 0 \wedge receivedSeqNum > 0$
  $\oplus SeqNum(receiver)(transferNode) \leq receivedSeqNum$
 then
  $\oplus TTL(receiver) := TTL(receiver) \mathbin{\lhd\!\!-}$
        $\{transferNode \mapsto receivedTTL - 1\}$
  $\oplus SeqNum(receiver) := SeqNum(receiver) \mathbin{\lhd\!\!-}$
        $\{transferNode \mapsto receivedSeqNum\}$
```

For the stabilize, we add two conditions:

(1) $nodes = \lambda x \cdot x \in Nodes | \mathrm{dom}(LinkStateTable(x))$,
(2) $\forall x, y \cdot x \in Nodes \wedge y \in nodes(x)$
  $\Rightarrow x \mapsto y \in closure(LinkStateTable(x))$,

to describe the scope of the link state propagation. We derive the stabilization property of the system as a theorem.

Theorem 1 *If the system is stable, and there exists a path in the network topology between node m and node n, and n is in the routing zone of m, then there exists a route from m to n in m's link state table. Let $Guards$ be the conjunction of all guards of* stabilize *event. Then, the formalization of the statement is*

$$\begin{aligned}&Guards \Rightarrow (\forall m, n \cdot m \mapsto n \in closure(NeighborLink)\\&\quad \wedge n \in \mathrm{ran}(LinkStateTable(m))\\&\quad \Rightarrow m \mapsto n \in closure(LinkStateTable(m))).\end{aligned}$$

By this refinement, every node has the up-to-date knowledge of its routing zone. It establishes the requirements **REQ-4** and **REQ-5**.

Fourth Refinement. To prevent a node from accessing other nodes' private information, we introduce $TransmittedLink$ to indicate the being transmitted link state information on links. We define $TransmittedSeqNum$ and $TransmittedTTL$ to specify the being transmitted sequence number and TTL value, respectively. A recipient may process the new received information of a node, while it does not send out the previous information of that node. So, we define a variable $Flag$ to denote whether a node can receive another node's link state information. Moreover, a receiver should ensure the connection with the sender without accessing the sender's private information ($inv5$). $inv6$ guarantees that the being transmitted link information for p are neighbor links of p.

```
$inv1 : TransmittedLink \in (Nodes \times Nodes) \rightarrow (Nodes \rightarrow$
        $(Nodes \leftrightarrow Nodes))$
$inv2 : TransmittedSeqNum \in (Nodes \times Nodes)$
        $\rightarrow (Nodes \rightarrow \mathbb{N})$
$inv3 : TransmittedTTL \in (Nodes \times Nodes) \rightarrow (Nodes \rightarrow \mathbb{N})$
$inv4 : Flag \in Nodes \rightarrow (Nodes \rightarrow \mathrm{BOOL})$
$inv5 : \forall m, n, p \cdot TransmittedLink(m \mapsto n)(p) \neq \emptyset \wedge n \neq p$
        $\Rightarrow m \mapsto n \in LinkStateTable(m)$
$inv6 : \forall m, n, p \cdot TransmittedLink(m \mapsto n)(p) \neq \emptyset \wedge n \neq p$
        $\Rightarrow \mathrm{dom}(TransmittedLink(m \mapsto n)(p)) = \{p\}$
```

If the connected links are down, then the being transmitted information on them is lost. So, we refine event obtainLinks to reset the being transmitted information on disconnected links. In addition, we adopt sendLinks, receiveLinks, discardLinks and cancelSendingLinks to model the distributed behavior along with the packet transmission. The sendLinks is enabled if the link information of $transferNode$ on $links$ is empty and the TTL value for $transferNode$ is greater than 0. Then, it puts the link state information, sequence number and the TTL on connected links, and sets the flag for $transferNode$ as TRUE to permit the reception about the link information of $transferNode$. Otherwise, the node just needs to reset the flag by cancelSendingLinks.

```
sendLinks
 **any** $sender\ transferNode\ links$
 **where**
  $sender \in Nodes$
  $transferNode \in Nodes$
  $transferNode \in \mathrm{dom}(LinkStateTable(sender))$
  $links = \{x \mapsto y | x = sender \wedge x \mapsto y \in LinkStateTable(x)\}$
  $\forall l \cdot l \in links \Rightarrow TransmittedLink(l)(transferNode) = \emptyset$
  $TTL(sender)(transferNode) > 0$
 **then**
  $TransmittedLink := TransmittedLink \mathbin{<\!\!+} (\lambda l \cdot l \in links|$
        $TransmittedLink(l) \mathbin{<\!\!+} \{transferNode \mapsto$
        $\{transferNode\} \lhd LinkStateTable(sender)\})$
  $TransmittedSeqNum := TransmittedSeqNum \mathbin{<\!\!+}$
        $(\lambda l \cdot l \in links|TransmittedSeqNum(l) \mathbin{<\!\!+} \{transferNode$
        $\mapsto SeqNum(sender)(transferNode)\})$
  $TransmittedTTL := TransmittedTTL \mathbin{<\!\!+} (\lambda l \cdot l \in links|$
        $TransmittedTTL(l) \mathbin{<\!\!+} \{transferNode \mapsto$
        $TTL(sender)(transferNode)\})$
  $Flag(sender) := Flag(sender) \mathbin{<\!\!+} \{transferNode \mapsto \mathrm{TRUE}\}$
```

By receiveLinks, refining transferLinks, the receiver deals with the link state information without consulting the private information of the sender. It obtains this information from the connected link $sender \mapsto receiver$. Clauses in the **with** part present the assignments for those received information. Both receiveLinks and discardLinks shall reset the being transmitted information on the link $sender \mapsto receiver$. Besides, receiveLinks changes the $Flag$ value for $transferNode$ from TRUE to FALSE, which forbids $receiver$ to receive the $transferNode$'s link information.

```
receiveLinks refines transferLinks
 **any** $sender\ receiver\ transferNode\ addlinks\ removelinks$
 **where**
  $\ominus sender \mapsto receiver \in LinkStateTable(sender)$
  $\ominus receivedlinks \in Nodes \leftrightarrow Nodes \wedge \mathrm{dom}(receivedlinks) =$
   $\{transferNode\}$
  $\ominus addlinks = receivedlinks$
  $\ominus SeqNum(receiver)(transferNode) \leq receivedSeqNum$
  $\ominus receivedTTL > 0 \wedge receivedSeqNum > 0$
  $\oplus addlinks =$
   $TransmittedLink(sender \mapsto receiver)(transferNode)$
  $\oplus TransmittedLink(sender \mapsto receiver)(transferNode) \neq \emptyset$
  $\oplus TransmittedTTL(sender \mapsto receiver)(transferNode) > 0$
   $\wedge TransmittedSeqNum(sender \mapsto receiver)(transferNode)$
   $> 0$
  $\oplus Flag(receiver)(transferNode) = \mathrm{TRUE}$
  $\oplus SeqNum(receiver)(transferNode) \leq$
   $TransmittedSeqNum(sender \mapsto receiver)(transferNode)$
 **with**
  $receivedlinks =$
   $TransmittedLink(sender \mapsto receiver)(transferNode)$
  $receivedTTL =$
   $TransmittedTTL(sender \mapsto receiver)(transferNode)$
  $receivedSeqNum =$
   $TransmittedSeqNum(sender \mapsto receiver)(transferNode)$
 **then**
  $\oplus TransmittedLink(sender \mapsto receiver) :=$
        $TransmittedLink(sender \mapsto receiver)$
        $\mathbin{<\!\!+} \{transferNode \mapsto \emptyset\}$
  $\oplus TransmittedSeqNum(sender \mapsto receiver) :=$
        $TransmittedSeqNum(sender \mapsto receiver)$
        $\mathbin{<\!\!+} \{transferNode \mapsto 0\}$
  $\oplus TransmittedTTL(sender \mapsto receiver) :=$
        $TransmittedTTL(sender \mapsto receiver)$
        $\mathbin{<\!\!+} \{transferNode \mapsto 0\}$
  $\oplus Flag(receiver) := Flag(receiver) \mathbin{<\!\!+}$
        $\{transferNode \mapsto \mathrm{FALSE}\}$
```

During the transmission of link state information, the being transmitted links, TTL values and sequence numbers maintain the consistency, described with $inv7$-$inv10$.

$$
\begin{array}{l}
inv7 : \forall m, n, p \cdot TransmittedLink(m \mapsto n)(p) \neq \emptyset \land n \neq p \\
\qquad \Rightarrow TransmittedTTL(m \mapsto n)(p) > 0 \\
inv8 : \forall l, p \cdot l \in Nodes \times Nodes \land TransmittedLink(l)(p) = \emptyset \\
\qquad \Rightarrow TransmittedTTL(l)(p) = 0 \\
inv9 : \forall m, n, p \cdot TransmittedLink(m \mapsto n)(p) \neq \emptyset \land n \neq p \\
\qquad \Rightarrow TransmittedSeqNum(m \mapsto n)(p) > 0 \\
inv10 : \forall l, p \cdot l \in Nodes \times Nodes \land TransmittedLink(l)(p) = \emptyset \\
\qquad \Rightarrow TransmittedSeqNum(m \mapsto n)(p) = 0
\end{array}
$$

4.3 Bordercasting-based route discovery

We have formalized the construction of routing zones in previous refinements. In the following steps, we focus on the development of the route request phase and the route reply phase in the changeable environment.

When we refer to the route request phase using bordercasting service, we pay more attention to these points:

1. the construction of bordercast trees with the aim of determining the intended forwarding nodes,
2. the records about the accumulated paths,
3. the routes selection with the destination in a node's routing zone,
4. the zone-based query control which steers a route request away from the covered regions.

The first and second points are developed in the next refinement. Other points are gradually formalized in the succeeding refinements.

Fifth Refinement. There are five new variables in this refinement. We define a variable $RouteRequest$, representing the set of route requests. A pair in $RouteRequest$ describes the source and the destination for a route request. For a route request, the source node and the destination node are distinct. The bordercast trees constructed by a node may vary from one route request to another. So, a bordercasting node needs to know the set of forwarding neighbors for a generated or received route request. $IntendedNeighbor$ specifies this information. The variable $AccumulatedPath$ keeps track of the accumulated paths along with requests delivery. Since we have considered the distributed behavior of the system, we define two variables to describe the corresponding being transmitted data. $TransmittedTag$ is used to determine that whether a neighbor is an intended receiver and the being transmitted accumulated routes are described by $TransmittedPath$.

$$
\begin{array}{l}
inv1 : RouteRequest \in Nodes \leftrightarrow Nodes \\
inv2 : IntendedNeighbor \in (Nodes \times (Nodes \times Nodes)) \\
\qquad \rightarrow \mathbb{P}(Nodes) \\
inv3 : AccumulatedPath \in Nodes \rightarrow ((Nodes \times Nodes) \\
\qquad \mathrel{+\!\!\!\rightarrow} (Nodes \leftrightarrow Nodes)) \\
inv4 : TransmittedTag \in (Nodes \times Nodes) \rightarrow \\
\qquad ((Nodes \times Nodes) \rightarrow \mathbb{N}) \\
inv5 : TransmittedPath \in (Nodes \times Nodes) \rightarrow \\
\qquad ((Nodes \times Nodes) \rightarrow (Nodes \leftrightarrow Nodes)) \\
inv6 : \forall s, d \cdot s \in Nodes \land d \in Nodes \land s \mapsto d \in RouteRequest \\
\qquad \Rightarrow s \neq d \\
inv7 : \forall s, n \cdot s \in Nodes \land n \in Nodes \land s \neq n \Rightarrow (\forall r \cdot s = prj1(r) \land \\
\qquad r \in \mathrm{dom}(AccumulatedPath(n)) \\
\qquad \Rightarrow r \in \mathrm{dom}(AccumulatedPath(s)))
\end{array}
$$

Even though we do not analyze the query control in this step, we keep in mind that a node shall not deal with a route request which has been processed.

The route request phase is issued when there is no route to the destination in the source node's routing table. Event sourceForwardRequest models this behavior. A parameter $nodes$ denotes the set of forwarding neighbors in the source's bordercast tree. Every node in $nodes$, an nonempty set, has at least one route to one of peripheral nodes within the source's routing zone. Then, the source places the request with an empty path on those determined links and indicates those intended recipients with the update $TransmittedTag$. Note that the source node and the destination node of a route request are distinct ($inv6$). An arbitrary route request which has been processed by some nodes is generated by the source ($inv7$).

```
sourceForwardRequest
 any $source\ destination\ request\ nodes\ links$
 where
   $source \in Nodes$
   $destination \in Nodes$
   $request = source \mapsto destination$
   $request \notin RouteRequest$
   $source \neq destination$
   $source \mapsto destination \notin closure(RoutingTable(source))$
   $nodes = \{q | source \mapsto q \in LinkStateTable(source) \land$
     $(\exists p, c \cdot c \subseteq LinkStateTable(source) \land q \mapsto p \in closure(c) \land$
     $\mathrm{card}(c) = zoneRadius - 1 \land$
     $(\forall s \cdot s \subseteq LinkStateTable(source) \land q \mapsto p \in closure(s)$
     $\land \mathrm{card}(s) \geq \mathrm{card}(c)))\}$
   $nodes \neq \emptyset$
   $links = \{p \mapsto q | p = source \land q \in nodes\}$
   $\forall l \cdot l \in links \Rightarrow TransmittedTag(l)(request) = 0$
   $request \notin \mathrm{dom}(AccumulatedPath(source))$
 then
   $RouteRequest := RouteRequest \cup \{request\}$
   $AccumulatedPath := AccumulatedPath \mathbin{\lhd\!\!-} \{source \mapsto$
         $(AccumulatedPath(source) \mathbin{\lhd\!\!-} \{request \mapsto \emptyset\})\}$
   $IntendedNeighbor := IntendedNeighbor \mathbin{\lhd\!\!-}$
         $\{(source \mapsto request) \mapsto nodes\}$
   $TransmittedTag := TransmittedTag \mathbin{\lhd\!\!-}$
         $(\lambda l \cdot l \in links | TransmittedTag(l) \mathbin{\lhd\!\!-} \{request \mapsto 1\})$
   $TransmittedPath := TransmittedPath \mathbin{\lhd\!\!-}$
         $(\lambda l \cdot l \in links | TransmittedPath(l) \mathbin{\lhd\!\!-} \{request \mapsto \emptyset\})$
```

receiveRequest states a receiver, an intended recipient, processes a received request. It records the accumulated route and resets the transmitted information on

$sender \mapsto receiver$. If the intended forwarding neighbors are nonempty, the node shall bordercast the request by bordercastRequest event. Both of them are abstract.

```
$inv8 : \forall n, m, r \cdot n \neq m \wedge TransmittedPath(n \mapsto m)(r) \neq \emptyset$
    $\Rightarrow r \in \mathrm{dom}(AccumulatedPath(n))$
$inv9 : \forall n, m, r, path \cdot n \neq m \wedge m \in IntendedNeighbor(n \mapsto r)$
    $\Rightarrow (r \mapsto path \in AccumulatedPath(n)$
    $\wedge TransmittedTag(n \mapsto m)(r) \neq 0$
    $\Rightarrow TransmittedPath(n \mapsto m)(r) = path)$
$inv10 : \forall n, m, r \cdot n \neq m \wedge TransmittedTag(n \mapsto m)(r) \neq 0$
    $\Rightarrow m \in IntendedNeighbor(n \mapsto r)$
$inv11 : \forall l, r \cdot l \in Nodes \times Nodes \wedge TransmittedTag(l)(r) = 0$
    $\Rightarrow TransmittedPath(l)(r) = \emptyset$
$inv12 : \forall n, r.n \in Node \wedge r \in RouteRequest \wedge r \in$
    $\mathrm{dom}(AccumulatedPath(n)) \Rightarrow AccumulatedPath(n)(r)$
    $\in Nodes \leftrightarrow Nodes$
$inv13 : \forall n, s, t \cdot n \in Nodes \wedge s \mapsto t \in Nodes \times Nodes \wedge$
    $s \mapsto t \in \mathrm{dom}(AccumulatedPath(n))$
    $\Rightarrow (\forall m, path \cdot path = AccumulatedPath(n)(s \mapsto t)$
    $\wedge m \in \mathrm{dom}(path) \Rightarrow \neg(m \mapsto m \in closure(path)))$
```

Here, we just show the receiveRequest since it shall be refined to more concrete events. The nodes receive a request from the transmitted information on links. So, if the being transmitted path on an arbitrary link is nonempty, then it indicates that the link source has received the request ($inv8$); in addition, the being transmitted path is equal to the accumulated route recorded in the link source for the request ($inv9$). A node is an intended neighbor for receiving a request from a link source if the being transmitted tag on that link is nonzero ($inv10$). For a request, the being transmitted path on the link is empty if the transmitted tag is 0 ($inv11$). It guarantees that the being transmitted information on links is consistent for requests. An accumulated path for a route request is a set of links ($inv12$). The discovered route is loop-free if the accumulated path is loop-free along with the requests propagation ($inv13$). $inv12$ and $inv13$ maintain the properties 2 and 3 in Definition 4. This refinement constitutes the system requirements **REQ-6** and **REQ-7**.

```
receiveRequest
 any $sender\ receiver\ request\ nodes\ path$
 where
  $sender \in Nodes \wedge receiver \in Nodes \wedge sender \neq receiver$
  $request \in RouteRequest$
  $TransmittedTag(sender \mapsto receiver)(request) = 1$
  $request \notin \mathrm{dom}(AccumulatedPath(receiver))$
  $nodes \in \mathbb{P}(Nodes)$
  $path \in Nodes \leftrightarrow Nodes \wedge sender \mapsto receiver \in path$
  $\forall n \cdot n \in \mathrm{dom}(path) \Rightarrow \neg n \mapsto n \in closure(path)$
 then
  $AccumulatedPath(receiver) :=$
      $AccumulatedPath(receiver) \mathbin{\lhd\!\!-} \{request \mapsto path\}$
  $IntendedNeighbor := IntendedNeighbor$
      $\mathbin{\lhd\!\!-} \{(receiver \mapsto request) \mapsto nodes\}$
  $TransmittedTag(sender \mapsto receiver) :=$
    $TransmittedTag(sender \mapsto receiver) \mathbin{\lhd\!\!-} \{request \mapsto 0\}$
  $TransmittedPath(sender \mapsto receiver) :=$
    $TransmittedPath(sender \mapsto receiver) \mathbin{\lhd\!\!-} \{request \mapsto \emptyset\}$
```

Sixth Refinement. In this refinement, we split receive Request into two events based on whether the receiver has valid routes to the destination in its routing zone.

One is receiveRequest_NoRoute that specifies the destination is not located in the receiver's routing zone. Here, the node requires to construct the bordercast tree with the calculated recipients denoting by $nodes$, and then forwards this request carrying the update $path$.

```
receiveRequest_NoRoute refines receiveRequest
 any $sender\ receiver\ request\ nodes\ path\ destination$
 where
  $\ominus nodes \in \mathbb{P}(Nodes)$
  $\ominus path \in Nodes \leftrightarrow Nodes \wedge sender \mapsto receiver \in path$
  $\oplus destination = prj2(request)$
  $\oplus receiver \mapsto destination \notin$
   $closure(LinkStateTable(receiver))$
  $\oplus nodes \subseteq \{q | receiver \mapsto q \in LinkStateTable(receiver) \wedge$
   $(\exists p, c \cdot c \subseteq LinkStateTable(receiver) \wedge q \mapsto p \in closure(c) \wedge$
   $\mathrm{card}(c) = zoneRadius - 1 \wedge$
   $(\forall s \cdot s \subseteq LinkStateTable(receiver) \wedge q \mapsto p \in closure(s) \wedge$
   $\mathrm{card}(s) \geq \mathrm{card}(c)))\}$
  $\oplus nodes \neq \emptyset$
  $\oplus path = TransmittedPath(sender \mapsto receiver)(request)$
   $\cup \{sender \mapsto receiver\}$
```

```
receiveRequest_HasRoute refines receiveRequest
 any $sender\ receiver\ request\ nodes\ path\ destination$
  $collection\ routes$
 where
  $\ominus path \in Nodes \leftrightarrow Nodes \wedge sender \mapsto receiver \in path$
  $\oplus nodes = \emptyset$
  $\oplus destination = prj2(request)$
  $\oplus receiver \mapsto destination \in$
   $closure(LinkStateTable(receiver))$
  $\oplus collection = \{S | S \subseteq LinkStateTable(receiver) \wedge$
   $\mathrm{card}(S) \leq zoneRadius \wedge$
   $receiver \mapsto destination \in closure(S)\}$
  $\oplus routes = \mathrm{union}(\{R | \forall S \cdot S \in collection \wedge$
   $R \in collection \wedge card(R) \leq card(S)\})$
  $\oplus path = TransmittedPath(sender \mapsto receiver)(request)$
   $\cup \{sender \mapsto receiver\} \cup routes$
```

The other one receiveRequest_HasRoute event creates a route reply with a discovered route. There exists at least one route to the destination in the receiver's routing zone. But the routes are not specified explicitly. So, the valid routes without useless links should be selected in terms of some metrics, e.g., the distance (hops). Therefore, we add two guards, for $collection$ and $routes$, to calculate the union of all shortest routes. card denotes the cardinality of an input finite set. Since it is unnecessary to bordercast this request, we set $nodes$ as the empty set.

We demonstrate that it indeed exists a route from the source to a node in the accumulated route for a request ($inv1$). So, the accumulated path, constructed by a recipient with the destination in its routing zone, contains the whole route from the source to the destination ($thm1$). This refinement establishes the requirement **REQ-8**.

```
$inv1 : \forall s, n \cdot s \in Nodes \wedge n \in Nodes \wedge s \neq n$
    $\Rightarrow (\forall r, path \cdot r \in RouteRequest \wedge$
    $r \mapsto path \in AccumulatedPath(n) \wedge s = prj1(r)$
    $\Rightarrow s \mapsto n \in closure(path))$
$thm1 : \forall s, t, n \cdot s \mapsto t \in RouteRequest \wedge n \in Nodes \wedge s \neq n$
    $\Rightarrow (\forall r, path \cdot r = s \mapsto t \wedge$
    $r \mapsto path \in AccumulatedPath(n)$
    $\wedge n \mapsto t \in closure(path) \Rightarrow s \mapsto t \in closure(path))$
```

Seventh Refinement. The purpose of the zone-based query control is to direct the route request away from the query source and the covered regions. In this step, we take into account the query control mechanism. A node has to know the coverage information within its routing zone when it builds a bordercast tree. Thus, we define a variable $ZoneCoverage \in Nodes \rightarrow ((Nodes \times Nodes) \rightarrow \mathbb{P}(Nodes))$ to describe this information.

A source node launches the route request and forwards it to the neighbors in its bordercast tree, so its routing zone is covered. Hence, we mark the routing zone as covered by refining sourceForwardRequest. $zone$ represents the set of nodes within the routing zone of $source$.

```
sourceForwardRequest refines sourceForwardRequest
 any $source\ destination\ request\ nodes\ links\ zone$
 where
  $\oplus zone = \{m | source \mapsto m \in$
   $closure(LinkStateTable(source)) \vee m = source\}$
  $\oplus source \notin ZoneCoverage(source)(request)$
 then
  $\oplus ZoneCoverage(source) := ZoneCoverage(source) \mathbin{<\!\!+}$
   $\{request \mapsto zone\}$
```

Moreover, a bordercaster's routing zone is marked as covered if it has delivered the received route request (refine bordercastRequest). We introduce a new event sendReply to mark the routing zone of a route discover, namely $node$ has an accumulated path from $source$ to $destination$, as covered.

```
sendReply
 any $node\ request\ source\ destination\ path\ zone$
 where
  $node \in Nodes$
  $request \in RouteRequest$
  $source = prj1(request) \wedge destination = prj2(request)$
  $request \in \mathrm{dom}(AccumulatedPath(node))$
  $path = AccumulatedPath(node)(request)$
  $source \mapsto destination \in closure(path)$
  $zone = \{q | node \mapsto q \in closure(LinkStateTable(node))$
   $\vee q = node\}$
 then
  $ZoneCoverage(node) := ZoneCoverage(node) \mathbin{<\!\!+}$
   $\{request \mapsto zone\}$
```

In order to prune the branches directed to covered peripheral nodes, we refine receiveRequest_NoRoute by adding two conditions for the calculation of $nodes$, the set of intended receiving nodes (the adding guard), : a) remove the previous bordercasting neighbor ($q \neq sender$); b) remove the neighbors directed to covered peripheral node(s). This refinement establishes the system requirement **REQ-9**.

```
receiveRequest_NoRoute refines receiveRequest_NoRoute
 any $sender\ receiver\ request\ nodes\ path\ destination$
 where
  $\ominus nodes \subseteq \{q | receiver \mapsto q \in LinkStateTable(receiver) \wedge$
   $(\exists p, c \cdot c \subseteq LinkStateTable(receiver) \wedge q \mapsto p \in closure(c)$
   $\wedge \mathrm{card}(c) = zoneRadius - 1 \wedge$
   $(\forall s \cdot s \subseteq LinkStateTable(receiver) \wedge q \mapsto p \in closure(s)$
   $\wedge \mathrm{card}(s) \geq \mathrm{card}(c)))\}$
  $\oplus nodes = \{q | receiver \mapsto q \in LinkStateTable(receiver) \wedge$
   $(\exists p, c \cdot c \subseteq LinkStateTable(receiver) \wedge q \mapsto p \in closure(c)$
   $\wedge \mathrm{card}(c) = zoneRadius - 1 \wedge$
   $(\forall s \cdot s \subseteq LinkStateTable(receiver) \wedge q \mapsto p \in closure(s)$
   $\wedge \mathrm{card}(s) \geq \mathrm{card}(c)) \wedge$
   $p \notin ZoneCoverage(receiver)(request)) \wedge q \neq sender\}$
```

4.4 Routing update

To formalize the routing update in detail, we define four variables. $ReplySender$ describes the previous senders of received replies. $ReplyPath$ specifies the received routes carried by route replies. In addition, we define *Transmitted Sender* and *TransmittedReplyPath* to represent the being transmitted information. Their relationship is described by $inv5$, i.e., for a route request, if the being transmitted discovered path on link $n \mapsto m$ is nonempty, then node n indeed has forwarded this reply to m. $inv6$ ensures that the received reply path contains the complete route from the source to the destination, i.e., the property 4 in the Definition 4.

```
$inv1 : ReplySender \in Nodes \rightarrow (Nodes \leftrightarrow (Nodes \times Nodes))$
$inv2 : ReplyPath \in Nodes \rightarrow (Nodes \times (Nodes \times Nodes) \rightarrow$
      $(Nodes \leftrightarrow Nodes))$
$inv3 : TransmittedSender \in (Nodes \times Nodes) \rightarrow (Nodes \leftrightarrow$
      $(Nodes \times Nodes))$
$inv4 : TransmittedReplyPath \in (Nodes \times Nodes) \rightarrow$
      $((Nodes \times Nodes) \rightarrow (Nodes \leftrightarrow Nodes))$
$inv5 : \forall n, m, r \cdot n \neq m \wedge TransmittedReplyPath(n \mapsto m)(r) \neq$
      $\emptyset \Rightarrow n \mapsto r \in TransmittedSender(n \mapsto m)$
$inv6 : \forall n, s, d, rq, path \cdot n \in Nodes \wedge s \in Nodes \wedge d \in Nodes \wedge$
      $rq = s \mapsto d \wedge path = ReplyPath(n)(s \mapsto rq) \wedge path \neq \emptyset$
      $\Rightarrow s \mapsto d \in closure(path)$
```

We refine the updateRoutingTable into two concrete events to show how nodes update the routing information on the basis of **REQ-10**. One is updateRoutingTable_IARP. It aims at updating the routing information according to the current connectivity within the local neighborhood. A node may have routes to the destinations beyond the routing zone in its routing table. So, we should identify the nodes that move out of the routing zone and then exclude their related routes. Parameters $addRoutes$ and $removeRoutes$ collect the links in the current routing zone and the links in the previous routing zone, respectively. The last guard states that there exists changes in the node's routing zone, which is an important condition for the updating.

```
updateRoutingTable_IARP refines updateRoutingTable
 any $node\ addRoutes\ removeRoutes\ zone\ nodes$
 where
  $\oplus zone = \{m | node \mapsto m \in closure(LinkStateTable(node))$
   $\vee m = node\}$
  $\oplus addRoutes = \{p, q \cdot p \mapsto q \in Nodes \times Nodes \wedge$
   $p \mapsto q \in LinkStateTable(node) | p \mapsto q\}$
  $\oplus nodes = \{m | node \mapsto m \notin closure(LinkStateTable(node))$
   $\wedge node \mapsto m \in closure(RoutingTable(node)) \wedge$
   $(\exists c \cdot c \subseteq RoutingTable(node) \wedge node \mapsto m \in closure(c) \wedge$
   $\mathrm{card}(c) \leq zoneRadius)\}$
  $\oplus removeRoutes = (nodes \cup zone) \lhd RoutingTable(node) \rhd$
   $(nodes \cup zone)$
  $\oplus nodes \neq \emptyset \vee \neg(addRoutes = zone \lhd RoutingTable(node) \rhd$
   $zone)$
```

By updateRoutingTable_Reply, we model the routing update for the node which receives a route reply packet. The nodes in one route are not necessary to record meaningless links in other routes. It satisfies the fact that a route reply shall be sent back to the query source by the reversed accumulated route. $path$ denotes the received entire route. Each

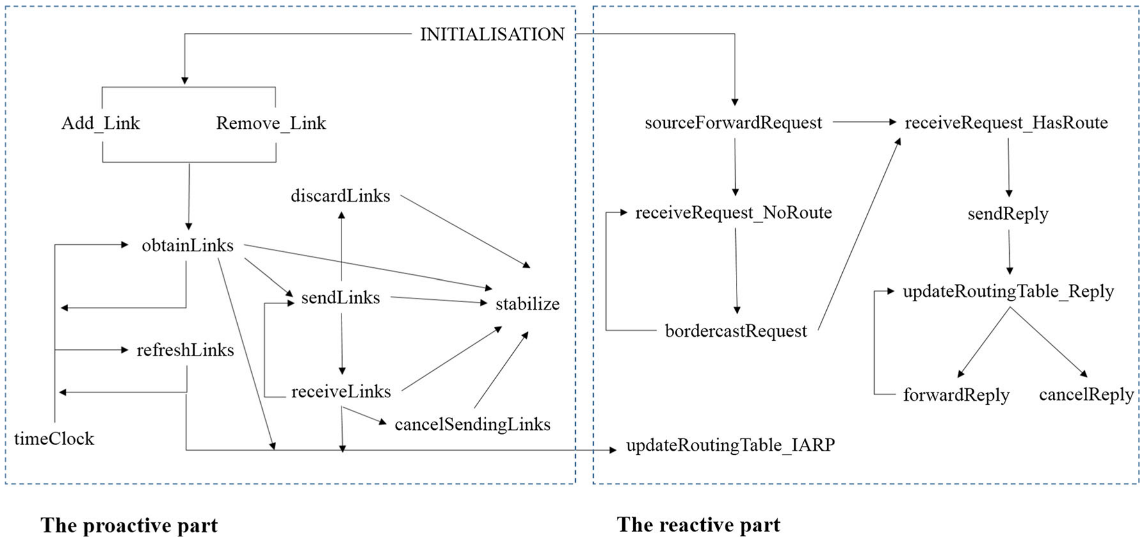

Fig. 2 Synchronization of the events in refinement 8

relaying node just needs to note down the route to the destination. So, we eliminate those unnecessary links from $path$, i.e., the route from the source to the current relaying node. $addRoutes$ denotes the result links, which are not contained in the routing table.

```
updateRoutingTable_Reply refines updateRoutingTable
 any $sender\ receiver\ addRoutes\ removeRoutes\ request\ path$
 where
   $\ominus node \in Nodes$
   $\ominus\neg(addRoutes = \emptyset \wedge removeRoutes = \emptyset)$
   $\oplus sender \in Nodes$
   $\oplus receiver \in Nodes$
   $\oplus\neg(addRoutes = \emptyset) \wedge removeRoutes = \emptyset$
   $\oplus sender \mapsto request \in$
   $TransmittedSender(sender \mapsto receiver)$
   $\oplus path \neq \emptyset \wedge path =$
   $TransmittedReplyPath(sender \mapsto receiver)(request)$
   $\oplus request \in closure(path)$
   $\oplus receiver \in (\mathrm{dom}(path) \setminus \{p | sender \mapsto p \in closure(path)$
   $\vee p = sender\})$
   $\oplus addRoutes = \{p | p \in \mathrm{dom}(path) \wedge$
    $p \mapsto receiver \in closure(path)\} \lhd path$
   $\oplus\neg addRoutes \subseteq RoutingTable(receiver)$
 with
   $node = receiver$
 then
   $\oplus ReplySender(receiver) := ReplySender(receiver) \cup$
        $\{sender \mapsto request\}$
   $\oplus ReplyPath(receiver) := ReplyPath(receiver) \Leftarrow$
        $\{(sender \mapsto request) \mapsto path\}$
   $\oplus TransmittedSender(sender \mapsto receiver) :=$
        $TransmittedSender(sender \mapsto receiver) \setminus$
        $\{sender \mapsto request\}$
   $\oplus TransmittedReplyPath(sender \mapsto receiver) :=$
        $TransmittedReplyPath(sender \mapsto receiver) \Leftarrow$
        $\{request \mapsto \emptyset\}$
```

We refine sendReply to start the route reply phase and introduce a new event forwardReply to forward the route reply to the next node determined by the accumulated path. A node shall check the connectivity to the next node before delivering a reply. In sendReply, $node$ sends the discovered path ($path$) to $next$ node, which is decided by $path$. Then it places $node$ and $path$ information on link $node \mapsto next$. Source node shall terminate the transmission of the corresponding reply with cancelReply event.

```
sendReply refines sendReply
 any $node\ request\ source\ destination\ path\ zone\ next$
 where
   $\oplus next \mapsto node \in AccumulatedPath(node)(request)$
   $\oplus node \mapsto next \in LinkStateTable(node)$
   $\oplus TransmittedReplyPath(node \mapsto next)(request) = \emptyset$
 then
   $\oplus TransmittedSender(node \mapsto next) :=$
        $TransmittedSender(node \mapsto next)$
        $\cup\{node \mapsto request\}$
   $\oplus TransmittedReplyPath(node \mapsto next) :=$
        $TransmittedReplyPath(node \mapsto next)$
        $\Leftarrow \{request \mapsto path\}$
```

Those events model the system requirement **REQ-10**. The entire formalization establishes requirements **REQ-1** and **REQ-2**. The synchronization of the developed events is shown in Fig. 2.

4.5 Model validation

We adopt the ProB, an animation and model checking tool, which is available for the Rodin, to validate our model. It allows us to perform the animation of Event-B models without translating those models into particular ones utilized in the ProB. Moreover, this animator supports stepwise animation of machines and non-deterministic operations. In Rodin, the ProB perspective presents a description of the current state of a machine, and a list of all enabled events, along with proper argument instantiations. Hence, the users can choose an available event with possible arguments to change the system state.

Since the nodes in the network, the broadcast period and zone radius are not explicitly defined in our formalization, there may be infinitely many system states. To avoid state space explosion problem and validate our model

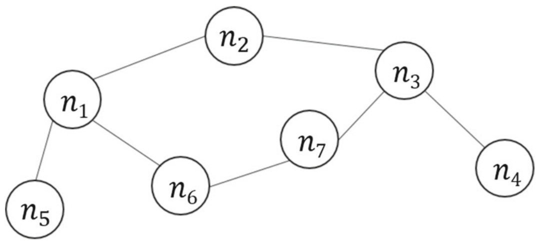

Fig. 3 The network topology for the model validation

properly, we introduce an auxiliary context to assign the defined set, $Nodes$, and some constants presented at follows. $partition(S, s_1, \ldots, s_n)$, a predicate, states that the sets $s_1, \ldots, s_n$ constitute partitions of the set S. Additionally, we set the argument $tick$ in the timeClock as 1, denoting that the time increment is 1.

$$\begin{array}{l}
axm1 : partition(Nodes, \{n1\}, \{n2\}, \{n3\}, \{n4\}, \{n5\}, \\
\quad \{n6\}, \{n7\}) \\
axm2 : lifetime = 4 \\
axm3 : period = 2 \\
axm4 : zoneRadius = 2
\end{array}$$

In our models, we define $closure$ function to compute the transitive closure of an input relation defined over $Nodes$. But this computation increases the difficulty of animation. To make the animation smooth, it requires to simplify this computation. So, we assume that the network topology, as presented in Fig. 3, is static. The variable $NeighborLink$ is considered as a constant for describing the connectivity of this network ($axm5$). Then, we utilize a constant $closureNL$ ($axm6$) to specify the transitive closure of $NeighborLink$ ($axm7$). Note that the system is not in the stable state if the stabilize event is not stimulated.

$$\begin{array}{l}
axm5 : partition(NeighborLink, \\
\quad \{n1 \mapsto n2, n1 \mapsto n5, n1 \mapsto n6\}, \\
\quad \{n2 \mapsto n1, n2 \mapsto n3\}, \{n3 \mapsto n2, n3 \mapsto n4, n3 \mapsto n7\}, \\
\quad \{n4 \mapsto n3\}, \{n5 \mapsto n1\}, \{n6 \mapsto n1, n6 \mapsto n7\}, \\
\quad \{n7 \mapsto n6, n7 \mapsto n3\}) \\
axm6 : closureNL \in Nodes \leftrightarrow Nodes \\
axm7 : partition(closureNL, NeighborLink, \\
\quad \{n1 \mapsto n3, n1 \mapsto n4, n1 \mapsto n7\}, \\
\quad \{n2 \mapsto n4, n2 \mapsto n5, n2 \mapsto n6, n2 \mapsto n7\}, \\
\quad \{n3 \mapsto n1, n3 \mapsto n5, n3 \mapsto n6\}, \\
\quad \{n4 \mapsto n1, n4 \mapsto n2, n4 \mapsto n5, n4 \mapsto n6, n4 \mapsto n7\}, \\
\quad \{n5 \mapsto n2, n5 \mapsto n3, n5 \mapsto n4, n5 \mapsto n6, n5 \mapsto n7\}, \\
\quad \{n6 \mapsto n2, n6 \mapsto n3, n6 \mapsto n4, n6 \mapsto n5\}, \\
\quad \{n7 \mapsto n1, n7 \mapsto n2, n7 \mapsto n4, n7 \mapsto n5\}) \\
axm8 : Request \in Nodes \leftrightarrow Nodes \\
axm9 : Request = \{n1 \mapsto n4\}
\end{array}$$

In terms of above network topology, we intend to discover a route from $n1$ to $n4$. We define $Request$ to store this route request with the source $n1$ and destination $n4$. We change the initial model by adding this auxiliary context and removing the existing machine, which describes the changes of the network topology. The subsequent refinements are formalized based on this context.

For the following refinements, we replace the predicates containing $closure$ as some other predicates in terms of the zone radius 2. For instance,

- $inv4$, in the third refinement, contains $m \mapsto n \in closure(LinkStateTable(m))$, indicating that n is in the current routing zone of m. So, we revise this part as $m \mapsto n \in LinkStateTable(m) \vee (\exists p \cdot m \mapsto p \in LinkStateTable(m) \wedge p \mapsto n \in LinkStateTable(m))$. We can replace some similar parts, such as sentence $source \mapsto m \in closure(LinkStateTable(source))$ for the setting of $zone$ in sourceForwardRequest, by the same way.
- One guard, $\forall x, y \cdot x \in Nodes \wedge y \in nodes(x) \Rightarrow x \mapsto y \in closure(LinkStateTable(x))$, for the stabilize in the third refinement, is revised as $\forall x, y \cdot x \in Nodes \wedge y \in nodes(x) \Rightarrow x \mapsto y \in LinkStateTable(x)$. Since the zone radius is 2, $nodes(x)$ represents the set of neighbors of x. To ensure there exists routes to those neighbors, it demands that $x \mapsto y$ is in $LinkStateTable(x)$.
- The guard of receiveRequest_HasRoute for $collection$, presented in the sixth refinement, is declared as $collection = \{S | S \subseteq LinkStateTable(receiver) \wedge card(S) \leq zoneRadius \wedge (receiver \mapsto destination \in S \vee (\exists p \cdot receiver \mapsto p \in S \wedge p \mapsto destination \in S))\}$. Based on the properties of $closure$, we can say that $receiver \mapsto destination \in closure(S)$.
- $nodes$ in receiveRequest_NoRoute shall be modified as $\{q | receiver \mapsto q \in LinkStateTable(receiver) \wedge (\exists p \cdot \neg(\exists x \cdot p \mapsto x \in LinkStateTable(receiver)) \wedge q \mapsto p \in LinkStateTable(receiver) \wedge p \notin ZoneCoverage(receiver)(request)) \wedge q \neq sender\}$, for the purpose of stating that q has a route to the peripheral node p.

Notice that those modifications just apply to the models with a particular zone radius 2. Some statements, containing the $closure$ function with an undetermined path, cannot be modified, such as $inv1$ and $thm1$ presented in the sixth refinement. Thus, we omit them and pay more attention to check their correctness manually when we perform the animation of our models.

The aim of our models is to find routes from $n1$ to $n4$. We carry out the animation by manually choosing some enabled operations. There exists routes $\{n1 \mapsto n2, n2 \mapsto n3, n3 \mapsto n4\}$ and $\{n1 \mapsto n6, n6 \mapsto n7, n7 \mapsto n3, n3 \mapsto n4\}$ in the network topology. They can be discovered by a sequence of operations. We present an example for the animation in Fig. 4. It lists the step-by-step operations, along with some parameters which improve the readability of the example, of the whole route discovery for the request $n1 \mapsto n4$. Moreover, it specifies the values of some variables after the corresponding operation(s). For instance, nodes $n1$, $n2$, $n3$, $n6$ and $n7$

1. timeClock(1) 2. timeClock(1)	$Time = 3$
3. obtainLinks$(n1, \{n1 \mapsto n2, n1 \mapsto n5, n1 \mapsto n6\}, \ldots)$ 4. obtainLinks$(n2, \{n2 \mapsto n1, n2 \mapsto n3\}, \ldots)$ 5. obtainLinks$(n3, \{n3 \mapsto n2, n3 \mapsto n4, n3 \mapsto n7\}, \ldots)$ 6. obtainLinks$(n6, \{n6 \mapsto n1, n6 \mapsto n7\}, \ldots)$ 7. obtainLinks$(n7, \{n7 \mapsto n3, n7 \mapsto n6\}, \ldots)$	$LinkStateTable = \{n1 \mapsto \{n1 \mapsto n2, n1 \mapsto n5, n1 \mapsto n6\},$ $n2 \mapsto \{n2 \mapsto n1, n2 \mapsto n3\},$ $n3 \mapsto \{n3 \mapsto n2, n3 \mapsto n4, n3 \mapsto n7\}, n4 \mapsto \emptyset, n5 \mapsto \emptyset,$ $n6 \mapsto \{n6 \mapsto n1, n6 \mapsto n7\}, n7 \mapsto \{n7 \mapsto n3, n7 \mapsto n6\}\}$
8. sendLinks$(n1, \{n1 \mapsto n2, n1 \mapsto n5, n1 \mapsto n6\}, \ldots)$ 9. receiveLinks$(n2, \{n1 \mapsto n2, n1 \mapsto n5, n1 \mapsto n6\}, \ldots)$ 10. receiveLinks$(n6, \{n1 \mapsto n2, n1 \mapsto n5, n1 \mapsto n6\},$ $\ldots)$	$LinkStateTable = \{n2 \mapsto \{n2 \mapsto n1, n2 \mapsto n3, n1 \mapsto n2, n1 \mapsto n5, n1 \mapsto n6\},$ $\ldots, n6 \mapsto \{n6 \mapsto n1, n6 \mapsto n7, n1 \mapsto n2, n1 \mapsto n5, n1 \mapsto n6\}, \ldots\}$
11. sendLinks$(n2, \{n2 \mapsto n1, n2 \mapsto n3\}, \ldots)$ 12. receiveLinks$(n1, \{n2 \mapsto n1, n2 \mapsto n3\}, \ldots)$ 13. receiveLinks$(n3, \{n2 \mapsto n1, n2 \mapsto n3\}, \ldots)$	$LinkStateTable = \{n1 \mapsto \{n1 \mapsto n2, n1 \mapsto n5, n1 \mapsto n6, n2 \mapsto n1, n2 \mapsto n3\},$ $\ldots, n3 \mapsto \{n3 \mapsto n2, n3 \mapsto n4, n3 \mapsto n7, n2 \mapsto n1, n2 \mapsto n3\}, \ldots\}$
14. sendLinks$(n3, \{n3 \mapsto n2, n3 \mapsto n4, n3 \mapsto n7\}, \ldots)$ 15. receiveLinks$(n2, \{n3 \mapsto n2, n3 \mapsto n4, n3 \mapsto n7\},$ $\ldots)$ 16. receiveLinks$(n7, \{n3 \mapsto n2, n3 \mapsto n4, n3 \mapsto n7\},$ $\ldots)$	$LinkStateTable = \{\ldots, n2 \mapsto \{n2 \mapsto n1, n2 \mapsto n3, n1 \mapsto n2, n1 \mapsto n5, n1 \mapsto n6,$ $n3 \mapsto n2, n3 \mapsto n4, n3 \mapsto n7\}, \ldots,$ $n7 \mapsto \{n7 \mapsto n3, n7 \mapsto n6, n3 \mapsto n2, n3 \mapsto n4, n3 \mapsto n7\}, \ldots\}$
17. sendLinks$(n6, \{n6 \mapsto n1, n6 \mapsto n7\}, \ldots)$ 18. receiveLinks$(n1, \{n6 \mapsto n1, n6 \mapsto n7\}, \ldots)$ 19. receiveLinks$(n7, \{n6 \mapsto n1, n6 \mapsto n7\}, \ldots)$	$LinkStateTable = \{n1 \mapsto \{n1 \mapsto n2, n1 \mapsto n5, n1 \mapsto n6, n2 \mapsto n1, n2 \mapsto n3,$ $n6 \mapsto n1, n6 \mapsto n7\}, \ldots, n7 \mapsto \{n7 \mapsto n3, n7 \mapsto n6, n3 \mapsto n2,$ $n3 \mapsto n4, n3 \mapsto n7, n6 \mapsto n1, n6 \mapsto n7\}, \ldots\}$
20. sendLinks$(n7, \{n7 \mapsto n3, n7 \mapsto n6\}, \ldots)$ 21. receiveLinks$(n3, \{n7 \mapsto n3, n7 \mapsto n6\}, \ldots)$ 22. receiveLinks$(n6, \{n7 \mapsto n3, n7 \mapsto n6\}, \ldots)$	$LinkStateTable = \{\ldots, n3 \mapsto \{n3 \mapsto n2, n3 \mapsto n4, n3 \mapsto n7, n2 \mapsto n1, n2 \mapsto n3,$ $n7 \mapsto n3, n7 \mapsto n6\}, \ldots, n6 \mapsto \{n6 \mapsto n1, n6 \mapsto n7, n1 \mapsto n2,$ $n1 \mapsto n5, n1 \mapsto n6, n7 \mapsto n3, n7 \mapsto n6\}, \ldots\}$
23. updateRoutingTable_IARP$(n1, \ldots)$ 24. updateRoutingTable_IARP$(n2, \ldots)$ 25. updateRoutingTable_IARP$(n3, \ldots)$ 26. updateRoutingTable_IARP$(n6, \ldots)$ 27. updateRoutingTable_IARP$(n7, \ldots)$	$RoutingTable = \{n1 \mapsto \{n1 \mapsto n2, n1 \mapsto n5, n1 \mapsto n6, n2 \mapsto n1, n2 \mapsto n3,$ $n6 \mapsto n1, n6 \mapsto n7\}, n2 \mapsto \{n2 \mapsto n1, n2 \mapsto n3, n1 \mapsto n2, n1 \mapsto n5,$ $n1 \mapsto n6, n3 \mapsto n2, n3 \mapsto n4, n3 \mapsto n7\}, n3 \mapsto \{n3 \mapsto n2, n3 \mapsto n4,$ $n3 \mapsto n7, n2 \mapsto n1, n2 \mapsto n3, n7 \mapsto n3, n7 \mapsto n6\},$ $n4 \mapsto \emptyset, n5 \mapsto \emptyset, n6 \mapsto \{n6 \mapsto n1, n6 \mapsto n7, n1 \mapsto n2, n1 \mapsto n5,$ $n1 \mapsto n6, n7 \mapsto n3, n7 \mapsto n6\},$ $n7 \mapsto \{n7 \mapsto n3, n7 \mapsto n6, n3 \mapsto n2, n3 \mapsto n4, n3 \mapsto n7,$ $n6 \mapsto n1, n6 \mapsto n7\}\}$
28. sourceForwardRequest$(n1, n1 \mapsto n4, \ldots)$	$AccumulatedPath = \{n1 \mapsto \{(n1 \mapsto n4) \mapsto \emptyset, \ldots\}, \ldots\}$ $IntendedNeighbor = \{(n1 \mapsto (n1 \mapsto n4)) \mapsto \{n2, n6\}, \ldots\}$
29. receiveRequest_HasRoute$(n2, n1 \mapsto n4, \ldots)$ 30. receiveRequest_NoRoute$(n6, n1 \mapsto n4, \ldots)$	$AccumulatedPath = \{n2 \mapsto \{(n1 \mapsto n4) \mapsto \{n1 \mapsto n2, n2 \mapsto n3, n3 \mapsto n4\}, \ldots\},$ $\ldots, n6 \mapsto \{(n1 \mapsto n4) \mapsto \{n1 \mapsto n6\}, \ldots\}, \ldots\}$ $IntendedNeighbor = \{\ldots, (n6 \mapsto (n1 \mapsto n4)) \mapsto \{n7\}, \ldots\}$
31. bordercastRequest$(n6, n1 \mapsto n4, \{n6 \mapsto n7\}, \ldots)$ 32. receiveRequest_HasRoute$(n7, n1 \mapsto n4, \ldots)$	$AccumulatedPath = \{\ldots, n7 \mapsto \{(n1 \mapsto n4) \mapsto \{n1 \mapsto n6, n6 \mapsto n7, n7 \mapsto n3,$ $n3 \mapsto n4\}, \ldots\}, \ldots\}$
33. sendReply$(n2, n1 \mapsto n4, \ldots)$ 34. sendReply$(n7, n1 \mapsto n4, \ldots)$	$TransmittedReplyPath = \{(n2 \mapsto n1) \mapsto \{(n1 \mapsto n4) \mapsto \{n1 \mapsto n2, n2 \mapsto n3,$ $n3 \mapsto n4\}, \ldots\}, \ldots,$ $(n7 \mapsto n6) \mapsto \{(n1 \mapsto n4) \mapsto \{n1 \mapsto n6, n6 \mapsto n7, n7 \mapsto n3,$ $n3 \mapsto n4\}, \ldots\}, \ldots\}$
35. updateRoutingTable_Reply$(n1, n1 \mapsto n4, \ldots)$ 36. updateRoutingTable_Reply$(n6, n1 \mapsto n4, \ldots)$	$RoutingTable = \{n1 \mapsto \{n1 \mapsto n2, n1 \mapsto n5, n1 \mapsto n6, n2 \mapsto n1, n2 \mapsto n3,$ $n6 \mapsto n1, n6 \mapsto n7, n3 \mapsto n4\}, \ldots, n6 \mapsto \{n6 \mapsto n1, n6 \mapsto n7,$ $n1 \mapsto n2, n1 \mapsto n5, n1 \mapsto n6, n7 \mapsto n6, n7 \mapsto n3, n3 \mapsto n4\}, \ldots\}$
37. forwardReply$(n6, n1 \mapsto n4, n1, \ldots)$ 38. updateRoutingTable_Reply$(n1, n1 \mapsto n4, \ldots)$	$RoutingTable = \{n1 \mapsto \{n1 \mapsto n2, n1 \mapsto n5, n1 \mapsto n6, n2 \mapsto n1, n2 \mapsto n3,$ $n6 \mapsto n1, n6 \mapsto n7, n3 \mapsto n4, n7 \mapsto n3\}, \ldots\}$

Fig. 4 An example for the animation of our model

obtain their neighbor information by step 3-7, and then the current value of $LinkStateTable$ is shown in the right side. We utilize "…" to denote the unchanged parts of variables. The invariants and axioms keep true during this animation.

5 Related work and conclusion

The purpose of ad hoc routing protocols is to find routes to deliver data packets. By far, there are many contributions about the formal verification of routing protocols. Wibling adopted model checkers, SPIN and UPPAAL, to verify a simplified version of the Lightweight Underlay Network Ad-hoc Routing Protocol(LUNAR) [24]. To avoid state space explosion problem, he analyzed some interesting topologies with a small number of nodes. Bhargavan utilized the HOL, an interactive prover, and SPIN to verify the properties of distance vector routing protocols [6]. With Isabelle/HOL, [7,25] presented the verification for the properties of AODV and DSR, respectively. Moreover, Hoang presented a formal development of the distributed topology discovery algorithm in Event-B [15]. He has proved that the system eventually reaches a stable state if the physical environment is inactive. We also formalize the system stabilization property. But we limit the view of a node to its routing zone rather than the whole network. Méry presented an incremental formal development of the DSR protocol with stepwise refinements in Event-B [19]. Note that they verified either a proactive protocol or a reactive protocol.

Our work is closely related to Méry's, but his analysis for the route discovery process is abstract (1) each node in a discovered route updates its routing table by adding the entire route. In addition, he did not formalize the route reply phase, (2) the forwarding nodes can check the private information of its neighbors, that is the distributed behavior of the system is not considered, (3) the situation that the being transmitted route requests may be lost owing to the link breakage is not modeled, (4) the validity and loop freedom of discovered routes are not stated. Therefore, his analysis is so abstract that it cannot well reflect the reality behaviors of the protocol. To make our model more reasonable and closer to specify the system, we take into account the system distributed behavior among messages propagation, and the route reply delivery process. Furthermore, our formalization guarantees the loop freedom and validity of those discovered routes.

In our work, we construct a formal specification of the ZRP, a hybrid routing framework, by a refinement-based method. To the best of our knowledge, we are the first to formally analyze a hybrid protocol. Our goal is to analyze the route discovery process on the basis of bordercasting service.

Under the dynamic network environment, each node delivers its neighbor information throughout its routing zone. The addressed issues are how to model the store forward process and how to control the link state propagation within a limited scope. Our model not only formally specifies these issues but also considers the stabilization property of the system. That is if the system is stable (the network environment is inactive for a long time), then each node has a route to the node within its routing zone when there exists a path between them in the network topology.

We also develop some invariants to verify the properties of the route discovery. Note that the goal of the ZRP is to discover the required routes by bordercasting service rather than broadcasting. According to Definition 4, we prove that each discovered route is validity and loop-free. Moreover, we validate our model with a particular instantiation of the system in Sect. 4.5.

Table 1 summarizes the statistics of discharged proof obligations. It has generated 401 proof obligations, and half of them are proved automatically. The rest of them, involving arithmetic or set operations, are manually proved. By discharging the generated proof obligations, we ensure that the refinements are correct and the properties (invariants) are preserved.

Table 1 Proof obligations statistics

Model	Total number of POs	Automatically discharged	Manually discharged
Initial Model	14	12	2
Refinement 1	15	13	2
Refinement 2	59	38	21
Refinement 3	30	16	14
Refinement 4	98	55	43
Refinement 5	79	38	41
Refinement 6	20	13	7
Refinement 7	20	14	6
Refinement 8	66	41	25
Total	401	240(60%)	161(40%)

Our formal analysis is helpful to verify extensions of the ZRP framework, such as the Independent Zone Routing (IZR) [23] framework which supports the configuration for the scope of each node's routing zone. The development also provides reference to analyze other hybrid routing protocols.

References

1. Abolhasan, M., Wysocki, T., Dutkiewicz, E.: A review of routing protocols for mobile ad hoc networks. Ad Hoc Netw. **2**(1), 1–22 (2004)
2. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (2005)
3. Abrial, J.R.: Modeling in Event-B System and Software Engineering. Cambridge University Press, Cambridge (2010)
4. 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. Int. J. Softw. Tools Technol. Transf. (STTT) **12**(6), 447–466 (2010)
5. 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, pp. 178–193. Springer, Berlin Heidelberg (2012). doi:10.1007/978-3-642-30885-7_13
6. Bhargavan, K., Obradovic, D., Gunter, C.A.: Formal verification of standards for distance vector routing protocols. J. ACM **49**(4), 538–576 (2002). doi:10.1145/581771.581775
7. Bourke, T., van Glabbeek, R., Höfner, P.: A mechanized proof of loop freedom of the (untimed) AODV routing protocol. In: Cassez, F., Raskin, J.F. (eds.) Automated Technology for Verification and Analysis, Lecture Notes in Computer Science. Springer, Berlin (2014). doi:10.1007/978-3-319-11936-6_5
8. Butler, M., Voisin, L., Muller, T.: Tooling. In: Romanovsky, A., Thomas, M. (eds.) Industrial Deployment of System Engineering Methods, pp. 157–185. Springer, Berlin Heidelberg (2013). doi:10.1007/978-3-642-33170-1_12
9. Cansell, D., Méry, D.: Formal and incremental construction of distributed algorithms: on the distributed reference counting algorithm. Theor. Comput. Sci. **364**(3), 318–337 (2006)
10. Clausen, T., Dearlove, C., Jacquet, P.: The optimized link state routing protocol version 2. draft-ietf-manet-olsrv2-00, Work in progress (2006)
11. Haas, Z.J., Pearlman, M.R., Samar, P.: The bordercast resolution protocol (BRP) for ad hoc networks. draft-ietf-manet-zone-brp-02.txt, IETF Internet Draft (2002)
12. Haas, Z.J., Pearlman, M.R., Samar, P.: The interzone routing protocol (IERP) for ad hoc networks. draft-ietf-manet-zone-ierp-02.txt, IETF Internet Draft (2002)
13. Haas, Z.J., Pearlman, M.R., Samar, P.: The intrazone routing protocol (IARP) for ad hoc networks. draft-ietf-manet-zone-iarp-02.txt, IETF Internet Draft (2002)
14. Haas, Z.J., Pearlman, M.R., Samar, P.: The zone routing protocol (ZRP) for ad hoc networks. draft-ietf-manet-zone-zrp-04.txt, IETF Internet Draft (2002)
15. Hoang, T., Kuruma, H., Basin, D., Abrial, J.R.: Developing topology discovery in Event-B. In: Leuschel, M., Wehrheim, H. (eds.) IFM, LNCS, pp. 1–19. Springer, Berlin Heidelberg (2009)
16. Johnson, D.B., Maltz, D.A.: Dynamic source routing in ad hoc wireless networks. In: Imielinski, T., Korth, H. (eds.) Mobile

Computing, pp. 153–181. Kluwer Academic Publishers, Dordrecht (1996)
17. Leuschel, M., Butler, M.: ProB: an automated analysis toolset for the B method. STTT **10**(2), 185–203 (2008)
18. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
19. Méry, D., Singh, N.K.: Analysis of DSR protocol in Event-B. In: Défago, X., Petit, F., Villain, V. (eds.) Stabilization, Safety, and Security of Distributed Systems, LNCS, vol. 6976, pp. 401–415. Springer, Berlin Heidelberg (2011)
20. Métayer, C., Voisin, L.: The Event-B mathematical language (2009). http://wiki.event-b.org/index.php/Event-B_Mathematical_Language
21. Perkins, C.E., Bhagwat, P.: Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. ACM SIGCOMM Comput. Commun. Rev. **24**, 234–244 (1994)
22. Perkins, C.E., Royer, E.M.: Ad-hoc on-demand distance vector routing. In: Mobile Computing Systems and Applications, 1999. Proceedings. WMCSA'99. Second IEEE Workshop, pp. 90–100. IEEE (1999)
23. Samar, P., Pearlman, M.R., Haas, Z.J.: Independent zone routing: an adaptive hybrid routing framework for ad hoc wireless networks. IEEE/ACM Trans. Network. (TON) **12**(4), 595–608 (2004)
24. Wibling, O., Parrow, J., Pears, A.: Automatized verification of ad hoc routing protocols. In: Formal Techniques for Networked and Distributed Systems, FORTE 2004, *LNCS*, vol. 3235, pp. 343–358. Springer (2004)
25. Yang, H., Zhang, X., Wang, Y.: A correctness proof of the dsr protocol. In: Cao, J., Stojmenovic, I., Jia, X., Das, S. (eds.) Mobile Ad-hoc and Sensor Networks, Lecture Notes in Computer Science, pp. 72–83. Springer, Berlin Heidelberg (2006). doi:10.1007/11943952_7