Developing Topology Discovery in Event-B*

Thai Son Hoang[1], Hironobu Kuruma[2], David Basin[1], and Jean-Raymond Abrial[1]

[1] Department of Computer Science, ETH Zurich
[2] Hitachi Systems Development Laboratory, Yokohama, Japan

Abstract. We present a formal development in Event-B of a distributed topology discovery algorithm. Distributed topology discovery is at the core of several routing algorithms and is the problem of each node in a network discovering and maintaining information on the network topology. One of the key challenges is specifying the problem itself. Our specification includes both safety properties, formalizing invariants that should hold in all system states, and liveness properties that characterize when the system reaches stable states. We establish these by appropriately combining proofs of invariant preservation, event refinement, event convergence, and deadlock freedom. The combination of these features is novel and should be useful for formalizing and developing other kinds of semi-reactive systems, which are systems that react to, but do not modify, their environment.

1 Introduction

We report here on a case study in critical system development using refinement. In our case study, we use the Event-B formalism [1] to specify and formally develop an algorithm for *topology discovery*, which is a problem arising in network routing. We proceed by constructing a series of models, where the initial models specify the system requirements and the final model describes the resulting system. We use the Rodin tool for Event-B [2] to prove that each successive model refines the previous one, whereby the resulting system is correct by construction.

The problem we examine is interesting for several reasons. First, it is a significant case study in specifying and developing distributed graph and routing algorithms. In routing protocols such as link-state routing [16], which is the basis for protocols such as OSPF [13, 12] and OLSR [14], every router in the network must build a graph representing the network topology. In this graph, the vertices represent routing nodes and there is an edge from node a to node b if a can directly transmit data to b. Each node uses this graph to determine the shortest path to all other nodes, from which it constructs its routing table, which describes the best next hop to each destination. The main challenge in topology discovery is to ensure that the distributed construction of these graphs, as well as their updates after network changes, proceeds correctly. While there has been some work on using model checkers and theorem provers to verify properties of routing protocols (e.g., [6]), there have been relatively few case studies (e.g., [1, 3, 15]) in using

* An extended version of this paper is [10] and a proof archive can be found at deploy-eprints.ecs.soton.ac.uk/31. Part of this research was carried out within the European Commission ICT project 214158 DEPLOY, www.deploy-project.eu/index.html. We thank Daniel Fischer, Matthias Schmalz, and Christoph Sprenger for their comments on drafts of this paper.

formal methods to develop such protocols. Our work provides some insights on how this can be done.

Second, as we will see, the problem of topology discovery is surprisingly nontrivial. The complexity is both in specifying the protocol's desired properties (what does it mean to "proceed correctly?") and in carrying out the development and proofs. This complexity comes from the fact that the protocol should function in dynamically changing environments. If we do not place constraints on the environment a priori (which we do not) then the actual topology may change faster than the nodes can propagate information about the changes they discover. For example, two nodes may be connected and not know it, but by the time they receive link information on this, they may be no longer connected. To address this, we present a novel approach to specifying and developing algorithms whose properties depend on the environment's dynamics. Our approach combines the use of *convergent events* in refinement (these are events that cannot take control of the system for ever) with a specification of deadlock freeness to specify the system's properties in stable system states.

Finally, our case study is representative of an important class of systems, which we call *(distributed) semi-reactive systems*. These are distributed systems where the environment is dynamically changing and, although the system cannot alter the environment, it must monitor and appropriately react to the changes in the environment. This includes, for example, distributed monitoring algorithms where the nodes must reach some kind of agreement about the environment's properties. Our approach suggests one way of developing systems in this general class.

2 Background on Event-B

Here we briefly describe the Event-B formalism; see [1, 4] for further details. A development is a set of models described by contexts and machines. Contexts specify a model's static part, in terms of sets, constants, and axioms, whereas machines specify the dynamic part and correspond semantically to transition systems. A machine has variables, defining its state, and an initial state. The possible states are constrained by invariants. State transitions are described by events, which are guarded commands, each consisting of a guard and an action. The guard is a conjunction of predicates formalizing the necessary condition under which an event may occur, and the action describes how the state variables change when the event occurs. Semantically, an event denotes a relation (v, v') between the pre-state v (before the event) and the post-state v' after the event. We will later refer to the pairs $v \mapsto v'$ as instances of an event.

Machine refinement provides a means to introduce details about the dynamic properties of a model. Event-B's theory of refinement is closely related to that of Action Systems [5]. In particular, a concrete machine can refine another abstract machine, whereby their states are related by a (simulation) relation called a gluing invariant. Refinement is used to develop systems that are correct by construction. One specifies a series of machines $M_0, M_1, \ldots, M_n$, where each M_{i+1} refines M_i, the initial machines formalize the system's requirements, and the final machines formalize the system itself.

We have used the *Rodin Tool* [2] to create and analyze Event-B models. This tool generates proof obligations that ensure the correctness of the systems developed. These

```
if DetectChange($x$,$v$) then
    UpdateLSDB($x$,$v$)
    UpdateSPFTree(LSDB)
    LSA ← CreateLSA($x$,$v$)
    Broadcast(LSA)
end if
□
if Receive(LSA) then
    if IsFresh(LSA) then
        UpdateLSDB(LSA)
        UpdateSPFTree(LSDB)
        Broadcast(LSA)
    else
        Drop(LSA)
    end if
end if
□
Broadcast(LSDB)
```

Fig. 1. Link-state algorithm for node v (loop body)

include: *invariant preservation* for establishing that invariants always hold; *refinement* between machines; and the *convergence* (termination) of sets of events (i.e., that the events in the set do not collectively diverge). Note that the convergence of some events cannot always be shown immediately and is then delayed to later refinements. In this case, the convergence of these events is *anticipated*.

3 Topology Discovery

In this section, we describe our requirements on the system and our assumptions on the environment for topology discovery. We begin by describing the problem and algorithm informally, in the context of link-state routing, which is one of its main applications.

3.1 Informal Description

Routing is the process of selecting paths through a network for sending data from a source to a destination. A path may require the data to travel over multiple hops, each hop being an intermediate router. At each router, data is forwarded using routing tables to select the next hop (the appropriate output port) based on the packet's destination address. It is the routing algorithm's task to build these routing tables. In link-state routing, this is done using several auxiliary data structures. In particular, each router maintains a link-state database (LSDB) that encodes its view of the topology of the communication network, i.e., the set of routers and the links between them. From its LSDB, a router computes a shortest path first (SPF) tree, using Dijkstra's algorithm [9]. The SPF tree is used to create the routing table: the next hop to some destination is simply the neighbor that constitutes the first link in the shortest path to that destination. Examples of routing algorithms that proceed this way include the Open Shortest Path First protocol (OSPF) [12, 13] and (optimized) link-state routing [7, 8].

In our case study, we focus on the important subproblem of *topology discovery*: discovering and maintaining local information about the network topology. This requires a distributed algorithm (protocol) since each node must construct its own local copy of the network topology. To do this, each node discovers changes in its own local communication environment and communicates them to other nodes. The nodes each individually build their own graphs, representing their local view of the global network topology.

To show how topology discovery is used in routing, Figure 1 presents a simplified view of link-state routing. The algorithm consists of an infinite loop, which runs on each node v. The loop's body nondeterministically chooses (represented by □) between three parts. From left-to-right these are: (1) detect and propagate changes; (2) receive and process changes; and (3) send information to neighboring nodes.

The first part describes how a node processes and propagates changes. Suppose a node v detects a change in the status of a link that joins some node x to v. The node v then adjusts its own link-state database (LSDB), which stores all topology graph nodes and edges. Afterwards, it updates its shortest path first (SPF) tree from the LSDB using Dijkstra's algorithm. Finally, it creates a link-state advertisement (LSA) describing the status (up or down) of the link from x to v, and starts flooding the network by broadcasting this to all of its neighbors. The second part describes a node's actions after receiving a link-state advertisement. If the LSA is fresh, then again the SPF tree is updated and the flooding is continued by sending the LSA to all neighbors. The third part states that a node v can, at anytime, start flooding the network by broadcasting information about its current link-state database. This can be implemented by v broadcasting an LSA describing the status of the link from x to y, for each pair of distinct nodes x and y. Alternatively, one message can be broadcast, describing the entire state of v's LSDB. In this case, the second part must be modified to also handle the reception of LSDBs.

These three parts implement basic link-state routing. If we are interested in pure topology discovery, it suffices to simply delete the two UpdateSPFTree statements. The resulting algorithm corresponds closely to what we will develop in Section 4.

A key point is the need for the third part, which initiates flooding even when no changes are present. This is required for two reasons. The first is to handle the possibility that LSAs are lost during communication, which can occur if a link goes down during message transit. The second reason is to handle the special case where disconnected parts of a network are reconnected. Suppose, for example, that the network is disconnected into two subnetworks S_1 and S_2, which each undergo changes and at some later time point become connected due to a link l coming up (i.e., l connects a node in S_1 with one in S_2). Just flooding both subnetworks with an LSA describing l being up is not enough for the nodes in S_1 to learn the topology of S_2 and vice versa. In actual link-state routing protocols, this third part, periodic flooding, occurs at fixed, relatively infrequent intervals. For example, in OSPF it takes place every 30 minutes.

Observe that the above algorithm description is still abstract and omits critical details. For example, nodes receive and propagate information at different times and hence a node may receive old LSAs containing invalid information about the network topology. How this is handled (e.g., using sequence numbers) and information is updated is not specified above. We must address precisely such details in our case study.

3.2 Requirements for Topology Discovery

As previously mentioned, it is surprisingly difficult to formulate the requirements for topology discovery. The protocol must operate in an *environment* where the status of links may change at any time. Moreover, the environment's behavior is out of the control of the protocol and not influenced by it (this is the notion of semi-reactive system, previously mentioned at the end of Section 1). If the environment changes sufficiently rapidly, then links reported as down may actually be up and vice versa. Hence the local LSDBs may bear little relationship to the actual network topology.

To tackle this problem, we focus on the limiting, and most important, case of the algorithm's behavior: its behavior when the environment is sufficiently quiescent. In this case, we expect that the local LSDBs will eventually converge (also called "stabilize" in the routing literature) to images of the actual global topology. Some care must be taken in precisely formalizing this, in particular to handle the previously mentioned problem that the network may not always be connected. In general, a node n can only learn about a link from a node a to its neighbor b when there is a path through the graph (representing the topology) from b to n.

Recall from basic graph theory that any graph can be decomposed into a collection of strongly-connected components. Our main system requirement is therefore:

System Requirement 1. If the environment is inactive for a sufficiently long time then for each strongly-connected component M, the local view (LSDB) of every node in M is in agreement with the actual topology, restricted to M.

Hence, when information about the system gained from link sensing (detecting communication neighbors) and communication stabilizes, each node has the correct view of the links between all nodes in its connected subnetwork.

We state one further requirement, which limits the possible local views of nodes during the protocol.

System Requirement 2. The local views of the nodes must be consistent with the past: a link listed as up is either up or was previously up.

This requirement rules out the case that a node concludes that a link is up that never was. So errors in the local topologies must effectively come from communication delays concerning status changes.

3.3 Environment Assumptions

Before developing a topology discovery algorithm, we must also be clear about our assumptions on the environment. We list them below.

Environment Assumption 1: There are only finitely many nodes.

Without this assumption, any notion of stability based on a hop-by-hop propagation of information would be unachievable.

Environment Assumption 2: There are directed, one-way links between some pairs of distinct nodes. Links may come up or go down at any time.

These links represent the ability to carry out directed (one-way) communication between two nodes. Links may be wired or wireless.

Environment Assumption 3: When there is a new link from node a to node b, then b is made aware of this. Likewise, when a link from a to b exists and is broken, b is also made aware of this.

We will refer to a link from a to b as either an *outward link* from a or an *inward link* to b. Assumption 3 reflects the ability to carry out "link sensing", whereby each node can sense its inward links. In practice, this must be realized by some kind of protocol, e.g., a must periodically announce its presence to b, or, in the bidirectional case, a handshake protocol initiated by b may be used. Note, as a result, that this assumption does not require that the receiver b immediately becomes aware of changes, but only eventually.

Environment Assumption 4: When a link goes down, any messages sent on it and not yet received are lost.

This reflects that communication is asynchronous. There is a delay (of unbounded length) between message transmission and reception, and messages can be lost during this time interval.

Environment Assumption 5: Nodes communicate by broadcasting whereby a may send a message to all b for which there exists a link from a to b.

Note that broadcasting is sufficient for topology discovery and is used during flooding. For other protocols, one might alternatively use point-to-point communication.

In the next section, we shall see how each of these requirements is formalized in the context of our Event-B development.

4 Formal Development

We now describe our development of topology discovery. We developed seven models. The initial models formalize our environmental assumptions and system requirements, whereas the subsequent models introduce design decisions for the resulting system.

Initial model specifies the protocol environment.
Refinement 1 introduces the observer event for observing stable states and adds system events to model how nodes update their link information.
Refinement 2 provides more details about link updates. Namely, a node updates information about its direct links or receives information about links from its neighbors.
Refinement 3 introduces sequence numbers for tracking fresh link-state information.
Refinement 4 uses message passing to transmit information about the status of links.
Refinement 5 separates the events into two sets: the set of events that update link-state information and those events that discard it as being redundant. The idea is to prove the convergence of the events that update the link-state information.
Refinements 6 introduces a variant for proving the convergence of some events.

Due to lack of space, we present below only selected parts of our formalization and omit proof details.

4.1 The Context and Initial Model

We begin by defining an Event-B context. In the context, we define the carrier set *NODES* of all network nodes and we axiomatize that it is finite. This formalizes **Environment Assumption 1**. Additionally, we define a (function) constant *closure* that, together with axioms, formalizes the transitive closure of binary relations between *NODES*. Note that ";" denotes forward relational composition.

axioms:
axm0_1 $\mathrm{finite}(\mathit{NODES})$
axm0_2 $\mathit{closure} \in (\mathit{NODES} \leftrightarrow \mathit{NODES}) \rightarrow (\mathit{NODES} \leftrightarrow \mathit{NODES})$
axm0_3 $\forall r \cdot r \subseteq \mathit{closure}(r)$
axm0_4 $\forall r \cdot \mathit{closure}(r); r \subseteq \mathit{closure}(r)$
axm0_5 $\forall r, s \cdot r \subseteq s \ \wedge\ s; r \subseteq s \ \Rightarrow\ \mathit{closure}(r) \subseteq s$

In our initial model, we formalize the behavior of the environment, where links (represented as pairs of nodes) may go up or down at any time. The variable $RLinks$ (R for real, i.e., actual links) represents the set of links that are currently up, whereas the variable $DLinks$ represents the set of links that have previously been up, but are now down. These sets are disjoint (**inv0_3**) since a link cannot be simultaneously both up and down. Note, however that we do not require that their union is the set of all links. This may be because two nodes are simply not communication neighbors or because their status has not yet been fixed. This set of "unknown" links is simply the complement of the set $RLinks \cup DLinks$.

variables: $RLinks, DLinks$

invariants:
- **inv0_1** $RLinks \in \mathit{NODES} \leftrightarrow \mathit{NODES}$
- **inv0_2** $DLinks \in \mathit{NODES} \leftrightarrow \mathit{NODES}$
- **inv0_3** $RLinks \cap DLinks = \varnothing$

Besides initializing $RLinks$ and $DLinks$ both to the empty set, there are two events: AddLink and RemoveLink. The former models that an arbitrary $link$ comes up. This link is then added to the set of $RLinks$ and removed from the set of $DLinks$ (if it is already there). The latter event handles the symmetric case. Note from the guards that if a link is in either set (i.e., its status is not unknown), then it has been up, at least once in the past.

```
AddLink
  any  $link$  where
    $link \notin RLinks$
  then
    $RLinks := RLinks \cup \{link\}$
    $DLinks := DLinks \setminus \{link\}$
  end
```

```
RemoveLink
  any  $link$  where
    $link \in RLinks$
  then
    $RLinks := RLinks \setminus \{link\}$
    $DLinks := DLinks \cup \{link\}$
  end
```

These events formalize **Environment Assumption 2**. Communication links are directed as the relations $RLinks$ and $DLinks$ are not necessarily symmetric.

4.2 The First Refinement

In our first refinement, we start to model the details of the protocol, although still very abstractly. In particular, we state that the link information stored at each nodes gets updated, although without yet specifying how.

We introduce two variables $rlinks$ and $dlinks$ with the following invariants. These two variables represent the current link-state information stored by each node.

invariants:
- **inv1_1** $rlinks \in \mathit{NODES} \rightarrow (\mathit{NODES} \leftrightarrow \mathit{NODES})$
- **inv1_2** $dlinks \in \mathit{NODES} \rightarrow (\mathit{NODES} \leftrightarrow \mathit{NODES})$
- **inv1_3** $\forall n \cdot rlinks(n) \subseteq RLinks \cup DLinks$
- **inv1_4** $\forall n \cdot dlinks(n) \subseteq RLinks \cup DLinks$
- **inv1_5** $\forall n \cdot rlinks(n) \cap dlinks(n) = \varnothing$

The first two invariants formalize that each node stores its own local information (a binary relation between *NODES*) about the status of links. Moreover, if a node has some information about a link, then this link must be either currently up or down (i.e., not unknown). This is represented by the invariants **inv1_3** and **inv1_4**. The last invariant, **inv1_5**, states that a node cannot store contradictory information about the same link. Of course, different nodes can have different information about the same link.

Note that, together with the events AddLink and RemoveLink from the initial model, these invariants establish **System Requirement 2**. We have that a link can be in $DLinks$ iff it is removed with RemoveLink iff it was previously added to $RLinks$ with AddLink (since no other events change the state of $RLinks$ and $DLinks$) and therefore was previously up. Hence a link is only in $RLinks \cup DLinks$ if it is up (left disjunct) or was previously up (right disjunct).

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

```
stabilize
  status  $ordinary$
  when
    $\forall x, y \cdot x \mapsto y \in RLinks \Leftrightarrow x \mapsto y \in rlinks(y)$
    $\forall x, y \cdot x \mapsto y \in DLinks \Leftrightarrow x \mapsto y \in dlinks(y)$
    $\forall m, n \cdot m \mapsto n \in closure(RLinks) \Rightarrow$
      $(\forall k \cdot (k \mapsto m \in rlinks(n) \Leftrightarrow k \mapsto m \in rlinks(m)) \wedge$
          $(k \mapsto m \in dlinks(n) \Leftrightarrow k \mapsto m \in dlinks(m)))$
  then  skip  end
```

The first two guards require that every node y knows the correct status of all its inward links, i.e., y has detected all environment changes with respect to its inward links. The last guard requires that if there is a path from a node m to n, then n has the same (up/down) information as m for all inward links to m. Hence, the observer event fires in those states where nodes know the correct status of their neighbors and this status has already been propagated through the network along all outward links. Intuitively, in stable states, all nodes have the maximum knowledge of the environment that can be acquired from link sensing and communication. We say that the *system is in a stable state* when the observer event can fire.[3]

A central property that we proved is the following.

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

We formulate this theorem in Event-B as follows, where $grdStabilize$ refers to the guard of the observer event.

[3] This notion of system stability is an instance of the general notion of a *stable system property* (see e.g., [11]), which is a property P of system states whereby if P is true of any reachable state s then P is true of all states reachable from s.

```
$grdStabilize$
   $\Rightarrow$  $(\forall M \cdot (\forall f, l \cdot f \in M \land l \in M \land f \neq l \Rightarrow f \mapsto l \in closure(RLinks))$
      $\Rightarrow$  $(\forall n \cdot n \in M$
            $\Rightarrow$  $M \lhd rlinks(n) \rhd M = M \lhd RLinks \rhd M \quad \land$
                $M \lhd dlinks(n) \rhd M = M \lhd DLinks \rhd M$ ))
```

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

We proved this theorem using the Rodin tool. The theorem itself constitutes part of the proof of **System Requirement 1**. Namely, in a stable state, each node has the correct view of all links in its strongly-connected component. It still remains to be proved that this stable state will be reached whenever the environment is inactive for a sufficient long time period. We prove this in Section 4.8.

In this model, we also introduce two new events, addlink and removelink, which modify the link-state information of some node.

```
addlink
   status  $anticipated$
   any  $n, link$  where
      $n \in NODES$
      $link \in RLinks \cup DLinks$
   then
      $rlinks(n) := rlinks(n) \cup \{link\}$
      $dlinks(n) := dlinks(n) \setminus \{link\}$
   end
```

```
removelink
   status  $anticipated$
   any  $n, link$  where
      $n \in NODES$
      $link \in RLinks \cup DLinks$
   then
      $rlinks(n) := rlinks(n) \setminus \{link\}$
      $dlinks(n) := dlinks(n) \cup \{link\}$
   end
```

The event addlink abstractly models a node receiving information on a link directly from the topology. Specifically, the event nondeterministically selects a node n and a link $link$ with a known status. It then updates n's local information about $link$, ensuring that it is added to the set of real (up) links and removed from the set of down links. Perhaps counterintuitively, the event may add a link to $rlinks(n)$ that is actually down, i.e., that belongs to $DLinks$. This reflects a key aspect of our distributed algorithm: the information nodes receive about the environment may be out-dated. As noted previously, being in $RLinks \cup DLinks$ simply means the node has been up in the past. But by the time n receives information that $link$ is up, the link may actually be down. The second event removelink is analogous. At this level of refinement, addlink and removelink are *anticipated*. That is, we delay the proof that these events converge to subsequent refinements.

From now on, we concentrate on the refinement of addlink. The refinement of removelink can be found in our on-line development archive.

4.3 The Second Refinement

In this refinement, we specify more concretely how link information is updated in each node. There are two cases. The first case models a direct update by the hello event. The second case models an indirect update by the transfer_rlink event.

```
hello
  refines  $addlink$
  status  $convergent$
  any  $n, m$  where
    $m \mapsto n \in RLinks$
    $m \mapsto n \notin rlinks(n)$
  then
    $rlinks(n) := rlinks(n) \cup \{m \mapsto n\}$
    $dlinks(n) := dlinks(n) \setminus \{m \mapsto n\}$
  end
```

```
transfer_rlink
  refines  $addlink$
  status  $anticipated$
  any  $n, m, x, y$  where
    $x \mapsto y \in rlinks(m) \cup dlinks(m)$
    $n \neq y$
  then
    $rlinks(n) := rlinks(n) \cup \{x \mapsto y\}$
    $dlinks(n) := dlinks(n) \setminus \{x \mapsto y\}$
  end
```

The event hello models a node n discovering information (e.g., by receiving a "hello" message) from a node m with an outward link to n. This event refines the abstract event addlink, where the abstract parameter $link$ is represented by the mapping $m \mapsto n$. The event transfer_rlink models a node n receiving information about a link $x \mapsto y$ from some node m, which is not necessarily a neighbor. The guard $n \neq y$ indicates that this is an indirect update, that is, $x \mapsto y$ is not an inward link of n. This refines the abstract event addlink, where the abstract parameter $link$ is represented by the mapping $x \mapsto y$.

The link-state information for down links is modeled analogously by the events goodbye and transfer_dlink, which are omitted here. Together, hello and goodbye formalize **Environment Assumption 3**.

At this stage, we also prove the convergence of the hello and goodbye events and we will prove the convergence of the transfer_rlink and transfer_dlink events in the next refinement, that is, they are anticipated at this point. By decomposing the convergence proof into different refinements we can simplify the proof by decomposing the events into two different subsets and then considering these subsets individually. Note that when proving the convergence, we still have the obligation of proving that the anticipated events do not increase the new variant. Taken together, these steps imply that the events reduce a composite variant, built from the lexicographic combination of the variants used in the two proofs.

We prove convergence by showing that these two events always decrease a *variant*, which is a set-valued expression. In this case the variant is

$$\{m \mapsto n \mid m \mapsto n \in RLinks \setminus rlinks(n)\} \cup$$
$$\{m \mapsto n \mid m \mapsto n \in DLinks \setminus dlinks(n)\} \ .$$

This is the set of inward links to n, where n has incorrect information. Informally, since the hello and goodbye events both provide correct information about one inward link of a node, they decrease the variant. Since the set of *NODES* is finite, this variant is also finite and thus well-founded.

4.4 The Third Refinement

In the next two refinement steps, we model communication between nodes. This is in contrast to the last step where nodes update their link information directly using the link information of other nodes, which is of course not realizable in a distributed system.

Before modeling communication, we first model how nodes track which information is fresh, i.e., whether the link information received is new or old. Namely, we introduce a new variable, $seqNum \in \mathit{NODES} \rightarrow (\mathit{NODES} \times \mathit{NODES} \rightarrow \mathbb{N})$ representing the sequence number stored at each node for each link. We omit listing here the invariants for $seqNum$. Moreover, to reason about the convergence of transfer_rlink and transfer_dlink, we introduce an auxiliary variable msg that "measures" the convergence of the event. This variable will not be used in the guards of the event, that is, it will not affect the execution of the events, hence we can safely remove this variable in the subsequent refinement.

In the initialization event, the sequence number for all links is set to 0 and msg is empty. The sequence number for a particular node and link first takes on a positive value after a direct update (e.g. in the hello event).

```
hello
  refines  $hello$
  any  $n, m$  where
    $m \mapsto n \in RLinks$
    $m \mapsto n \notin rlinks(n)$
  then
    $rlinks(n) := rlinks(n) \cup \{m \mapsto n\}$
    $dlinks(n) := dlinks(n) \setminus \{m \mapsto n\}$
    $seqNum(n) := seqNum(n) \lhd\!\!- \{(m \mapsto n) \mapsto seqNum(n)(m \mapsto n) + 1\}$
    $msg := msg \cup (\{m \mapsto n \mapsto seqNum(n)(m \mapsto n) + 1\} \times (\mathit{NODES} \setminus \{n\}))$
  end
```

The only difference with the abstract version is the last two assignments, which increment the sequence number ($\lhd\!\!-$ denotes relation overriding) and update msg. Since the event's guard is unchanged and the additional assignment modifies only a new variable, this clearly refines the corresponding abstract hello event. Once new information is detected by n, this information must be propagated to all the other nodes in the network.

For indirect updates, the sequence number for a particular link is not updated, but simply passed from one node to another.

```
transfer_rlink
  refines  $transfer\_rlink$
  status  $convergent$
  any  $n, m, x, y, sn$  where
    $m \mapsto n \in RLinks$
    $sn \leq seqNum(m)(x \mapsto y)$
    $seqNum(n)(x \mapsto y) < sn$
    $\forall k \cdot seqNum(k)(x \mapsto y) = sn \Rightarrow x \mapsto y \in rlinks(k)$
  then
    $rlinks(n) := rlinks(n) \cup \{x \mapsto y\}$
    $dlinks(n) := dlinks(n) \setminus \{x \mapsto y\}$
    $seqNum(n) := seqNum(n) \lhd\!\!- \{(x \mapsto y) \mapsto sn\}$
    $msg := msg \setminus \{x \mapsto y \mapsto sn \mapsto n\}$
  end
```

Compared to the abstract version of the event, there is an additional parameter, sn, for the sequence number associated with the link-state information. This sequence number sn is no more than the sequence number that m has for the same link. The reason is that the original message came from m and sequence numbers are never decreased.[4] The sequence number sn is (strictly) greater than n's sequence number for the same link, that is, n only updates its local state with new information. The last guard states that for any node k with the same sequence number for the same link $x \mapsto y$, that link is in the set of up links for k. This ensures that there will be no conflicting information in the network. Note that this guard cheats in the sense that it cannot be directly implemented. This cheating will be eliminated in a subsequent refinement. The additional assignments in the event's action, with respect to the abstract version, update n's sequence number for the link $x \mapsto y$ and remove this information from the set msg.

We also proved the convergence of the transfer_rlink and transfer_dlink events. The variant is just msg. This, together with the convergence proof from the last refinement, shows that the events hello, goodbye, transfer_rlink, transfer_dlink decrease a combined lexicographic variant.

The guard of the observer event stabilize (from the first refinement) is also refined using information about sequence numbers. It becomes:

```
stabilize
  when
    $\forall x, y \cdot x \mapsto y \in RLinks \Leftrightarrow x \mapsto y \in rlinks(y)$
    $\forall x, y \cdot x \mapsto y \in DLinks \Leftrightarrow x \mapsto y \in dlinks(y)$
    $\forall n1, n2, link \cdot n1 \mapsto n2 \in RLinks \Rightarrow$
      $seqNum(n1)(link) \leq seqNum(n2)(link)$
  then  skip  end
```

The first two guards are unchanged. What is new is the last guard, which states that for any pair of nodes $n1$ and $n2$, and link $link$, if $n1$ has a direct communication link to $n2$, then $n2$'s information about $link$ is not older than $n1$'s.

4.5 The Fourth Refinement

We now model communication. We first remove the auxiliary variable msg. We also remove the assignments that modify msg from the events hello and goodbye. We then introduce three variables: $SChan$ of type $(NODES \times NODES) \rightarrow ((NODES \times NODES) \rightarrow \mathbb{N})$ and $RChan$ and $DChan$, both of type $(NODES \times NODES) \rightarrow (NODES \leftrightarrow NODES)$. For each pair of nodes, the link-state information is a relation between *NODES*, formalizing the set of pairs of nodes on the communication channel. For all nodes m and n, $RChan(m \mapsto n)$ (respectively, $DChan(m \mapsto n)$) is the set of up (down) link information that is transferred from m to n. The channel $SChan$ associates sequence numbers to the links in the link-state channels. Thus $SChan(m \mapsto n)$ stores information about the sequence numbers that are in transit from m to n.

[4] However, sn can differ from m's sequence number, since during the time for the message to reach n, m can in the meantime update its sequence number for the same link.

Communication between nodes uses the above channels, so the abstract events for transferring link information (namely, transfer_rlink and transfer_dlink) must each be split into a pair of events for sending and receiving information. The following diagram illustrates what happens. First, the node m *sends* the information to the channels and afterwards the node n *receives* information from the channels. In our development, each transfer event is refined by a receive event and we add a new send event, which therefore refines skip. In our diagram, the top part is the abstraction (skip and *transfer*) and the bottom part is the refinement (i.e., *send* and *receive*).

$$\begin{tikzcd}
\text{m} \arrow[r, "\mathsf{skip}"] & {} \arrow[r, "\textit{transfer}"] & \text{n} \\
\text{m} \arrow[r, "\textit{send}"] & \boxed{\text{channels}} \arrow[r, "\textit{receive}"] & \text{n}
\end{tikzcd}$$

Below is the description of the new event for sending information about an up link from m to n.

```
send_rlink
  status  $anticipated$
  any  $m, n, link$  where
    $m \mapsto n \in RLinks$
    $SChan(m \mapsto n)(link) = 0$
    $link \in rlinks(m)$
  then
    $SChan(m \mapsto n) := SChan(m \mapsto n) \lhd\!\!\!- \{link \mapsto seqNum(m)(link)\}$
    $RChan(m \mapsto n) := RChan(m \mapsto n) \cup \{link\}$
  end
```

For a node to send information about certain $link$, this event requires that the information about the same $link$ from the last send has been received. This is formalized by the guard stating that the corresponding sequence number in the channel is 0. The information is then sent by placing it on the outward links from m to n. The guard $m \mapsto n \in RLinks$ (i.e. the link from m to n is currently up) formalizes **Environment Assumption 5**.

The abstract transfer_rlink is refined to specify the following event receive_rlink.

```
receive_rlink
  refines  $transfer\_rlink$
  any  $m, n, x, y$  where
    $seqNum(n)(x \mapsto y) < SChan(m \mapsto n)(x \mapsto y)$
    $x \mapsto y \in RChan(m \mapsto n)$
  then
    $rlinks(n) := rlinks(n) \cup \{x \mapsto y\}$
    $dlinks(n) := dlinks(n) \setminus \{x \mapsto y\}$
    $seqNum(n) := seqNum(n) \lhd\!\!\!- \{(m \mapsto n) \mapsto SChan(m \mapsto n)(x \mapsto y)\}$
    $SChan(m \mapsto n) := SChan(m \mapsto n) \lhd\!\!\!- \{(x \mapsto y) \mapsto 0\}$
    $RChan(m \mapsto n) := RChan(m \mapsto n) \setminus \{x \mapsto y\}$
  end
```

The link-state information is retrieved from the channels from m to n. Here, the abstract parameter sn is refined as $SChan(m \mapsto n)(x \mapsto y)$. The refinement of transfer_dlink to receive_dlink is analogous.

Note that the event receive_rlink receives only genuinely new messages. Hence it is necessary to introduce a *complement* event that discards obsolete information, both for up and down links. Another reason for introducing discard events is that, without them, we would not be able to prove the deadlock freeness property in the next refinement level. Below is the event for discarding information about an up link (the new event discard_dlink is analogous).

```
discard_rlink
  **status**  $anticipated$
  **any**  $m, n, link$  **where**
    $SChan(m \mapsto n)(link) \leq seqNum(n)(link)$
    $link \in RChan(m \mapsto n)$
  **then**
    $SChan(m \mapsto n) := SChan(m \mapsto n) \Leftarrow \{link \mapsto 0\}$
    $RChan(m \mapsto n) := RChan(m \mapsto n) \setminus \{link\}$
  **end**
```

The link-state information is obsolete since the node has already received more recent information about $link$ in the channel. Hence, the information is simply discarded from the channel. This new event refines skip since the actions only effect the new variables, $SChan$ and $RChan$.

Now that we have explicitly introduced communication, we refine the environment event RemoveLink to account for **Environment Assumption 4**. That is, when a link goes down, any messages sent on it and not yet received are lost.

```
RemoveLink
  **refines**  $RemoveLink$
  **any**  $link$  **where**
    $link \in RLinks$
  **then**
    $RLinks := RLinks \setminus \{link\}$
    $DLinks := DLinks \cup \{link\}$
    $SChan := SChan \Leftarrow (\{link\} \times \{NODES \times NODES \times \{0\}\})$
    $RChan(link) := \varnothing$
    $DChan(link) := \varnothing$
  **end**
```

This trivially refines the abstract RemoveLink event since the guard is unchanged and the new assignments only modify new variables.

Note that at this point all the events can be straightforwardly implemented in a distributed system. That is, the events no longer "cheat" and perform tests or actions that would not be algorithmically realizable.

4.6 The Fifth Refinement

Our machine in the fourth refinement constitutes a (high-level) protocol implementation. However, we have not yet established the convergence of the events send_rlink and discard_rlink (and correspondingly for $dlink$). There is a good reason for this: these events do not converge and should not converge. As we saw in Figure 1 (third part), each node periodically broadcasts information about its LSDB and its neighbors repeatedly receive this information, even when it is not new. What we prove then is that the system eventually does reach a stable state (assuming that the environment does not change), despite continually broadcasting and receiving redundant information.

To prove this, we shall partition these four non-convergent events each into two parts: a convergent and divergent part. We accomplish this by defining a restricted local notion of stability, called neighbor stability, and showing that the neighbor-stable parts diverge and, conversely, the neighbor-unstable parts converge.

Given a link $link$ and a link from m to n, we say the information about $link$ is *neighbor stable* from m to n if n's sequence number for $link$ is at least as large as m's. This means that the information about $link$ in m does not need to propagate to n and therefore further information coming from m about $link$ will not change this neighbor-stable status. Using this notion, we can restate the third guard of the observe event stabilize (from Section 4.4) as follows: Any $link$ is neighbor-stable for any up link from m to n.

We now partition the events by adding either the guard $seqNum(m)(link) \leq seqNum(n)(link)$ or its complement. For example, we partition send_rlink into the two events send_rlink_stable and send_rlink_unstable. For send_rlink_stable we add the above guard and for send_rlink_unstable we add the complement as a guard. We partition the other three events discard_rlink, send_dlink, and discard_dlink similarly. Note that we must partition the discard events as information must also be discarded in neighbor-unstable states. The reason for this is that communication is asynchronous and therefore information may be sent in a stable state but received in an unstable state.

Given this partition, we prove the convergence of the events send_rlink_unstable and send_dlink_unstable using the variant

$$\{m \mapsto n \mapsto link \mid SChan(m \mapsto n)(link) \leq seqNum(n)(link)\}\,.$$

This denotes the set of old messages on all channels. We will prove the convergence of discard_rlink_unstable and discard_dlink_unstable in the next refinement level and hence they act as anticipated events here.

In this refinement step, we also proved the following theorem about the deadlock freeness of a set of events. Namely, the guard of the event stabilize is equivalent to the negation of the disjunction of the guards of the following eight events: hello, goodbye, send_rlink_unstable, send_dlink_unstable, receive_rlink, discard_rlink_unstable, receive_dlink, and discard_dlink_unstable. Hence, if none of these eight events is enabled, then stabilize is enabled and the system is therefore in a stable state.

Moreover, we also proved theorems stating that the four events send_rlink_stable, send_dlink_stable, discard_rlink_stable, and discard_dlink_stable maintain the system's stable state, that is, if we assume that the state before the event execution is stable, we have to prove that the state after the event execution is also stable. However, $stable$

refers to $RLinks$, $DLinks$,$rlinks$, $dlinks$, and $seqNum$ only, whereas our events (send_rlink_stable, send_dlink_stable, discard_rlink_stable, and discard_dlink_stable) only modify the information in the channels, i.e., $SChan$, $RChan$, and $DChan$, so the above events will maintain the stable state.

4.7 Sixth Refinement

In this refinement step, we prove the convergence of the discard_rlink_unstable and discard_dlink_unstable using the variant

$$\{m \mapsto n \mapsto link \mid SChan(m \mapsto n)(link) \neq 0\} \cap$$
$$\{m \mapsto n \mapsto link \mid seqNum(n)(link) < seqNum(m)(link)\}\,.$$

4.8 Partial Convergence implies Stability

In contrast to the development of terminating programs, we now only prove the convergence of a subset of the events. Nevertheless, we show that this is adequate to establish **System Requirement 1**. Namely, if the environment is inactive for a sufficiently long time, then for each strongly-connected component M, the local view of every node in M is in agreement with the actual topology, restricted to M.

First, we introduce the notion of a *run* of Event-B together with a strong-fairness assumption. A run of an Event-B model is an infinite sequence of states obtained from an initial state by executing events of the model. We call a run *strongly fair* with respect to a set of events E if it respects the following *strong-fairness* assumption with respect to E: if an event from E is enabled infinitely often, then it will be taken infinitely often. This assumption will hold for any reasonable implementation of topology discovery.

At the last refinement, the set of events can be divided into the following groups.

1. A set of environment events $Env = \{Env_1, \ldots, Env_l\}$. In our case, there are just the two events AddLink and RemoveLink.
2. An observer event Obs. This observer event has skip as its action and its guard specified that the system is in stable state. Hence it is of the form:

   when *stable* **then** skip **end**

   In our development, this is the stabilize event.
3. A set of convergent events $CE = \{CE_1, \ldots, CE_m\}$. In our development, the convergent events are hello, goodbye, send_rlink_unstable, send_dlink_unstable, receive_rlink, discard_rlink_unstable, receive_dlink, and discard_dlink_unstable.
4. A set of divergent events $DE = \{DE_1, \ldots, DE_n\}$. These events are send_rlink_stable, send_dlink_stable, discard_rlink_stable, and discard_dlink_stable.

We will now prove the following theorem:

Theorem 2 (System Stabilizes). *Assume that the following propositions hold:*

i) Deadlock-freedom for the observer event Obs and convergent events CE. In particular,

$$stable \Leftrightarrow \neg(Grd(CE_1) \vee \cdots \vee Grd(CE_m))\,.$$

ii) The events in CE converge using a well-founded variant V.
iii) The events in DE do not increase V.
iv) The events in DE preserve $stable$. By this we mean that none of the DE events disable the guard of Obs*.*
v) The events in CE are strongly fair.

Then if the environment is eventually quiescent (i.e., at some point no environment events $Env_1, \ldots, Env_l$ from the first group occur) then the system will eventually reach a stable state and remain in this state.

In our case, we are assuming Proposition (v), and the other propositions have already been previously proved.[5] Our proof of Theorem 2 is by contradiction and proceeds as follows. Assume that there is a strongly fair run R with a quiescent suffix, but which never reaches a stable state. Then there must be infinitely many i such that $R(i)$ does not satisfy "stable". Let r be a quiescent suffix of R. By Proposition (i), there are infinitely many states such that some event in CE is enabled. By the fairness assumption, Proposition (v), the events in CE must be taken infinitely often on r. Since there are no environment events and by Proposition (ii) all events in CE decrease the variant, whereas by Proposition (iii), other system events (i.e., Obs and DE) do not increase the variant V, the variant V decrease infinitely often in r. This contradicts the well-foundedness of V. Therefore, all strongly fair runs with a quiescent suffix eventually reach a stable state. Moreover, once in a stable state, all the events in CE are disabled and, by Proposition (iv), the events in DE preserve the stable state. Together with the fact that event Obs does not change the state (its action is skip), it follows that the system stays in the stable state. This concludes our proof. Note that this proof is a traditional "paper and pencil proof", rather than a proof using the Rodin tool.

The system referred to in the theorem statement is the machine M_5 given by the 5th refinement, rather than the machine M_4 from the 4th refinement, which is our implementation. However, M_5 simply partitions four of M_4's events. Therefore the proof of Theorem 2 just given for M_5 can be naturally mapped to M_4. Namely, the partition of M_4's events into stable and unstable events in M_5 gives rise to a partition of their instances. Therefore Theorem 2 also holds for M_4 if we restate the fairness assumption in Theorem 2 as follows: "If an instance of event is enabled infinitely often, then it will be taken infinitely often."

Finally, recall Theorem 1, proved in Section 4.2, which states that in a stable state, each node has the correct view of all links in its strongly-connected component. It follows from this and Theorem 2 that the system M_4 satisfies **System Requirement 1**.

4.9 Summary — Proof Statistics

In Table 1 we give proof statistics of the development in the Rodin Tool. These statistics measure the size of the model, the proof obligations generated and discharged by the Rodin Platform, and those interactively proved. Note that there are many proof obligations in the 4th refinement due to the introduction of three different channels. In order to

[5] We proved Propositions (i) and (iv) in the 5th refinement and proved Propositions (ii) and (iii) in the 2nd, 3rd, 5th, and 6th refinements.

Model	Number of Proof Obligations	Automatically Discharged	Interactively Discharged
Context	3	0(0%)	3(100%)
Initial Model	9	9(100%)	0(0%)
1st Refinement	31	26(84%)	5(16%)
2nd Refinement	30	23(77%)	7(23%)
3rd Refinement	74	37(50%)	37(50%
4th Refinement	159	79(50%)	80(50%)
5th Refinement	44	7(16%)	37(84%)
6th Refinement	8	0(0%)	8(100%)
Total	358	181(51%)	177(49%)

Table 1. Proof statistics

guarantee the correctness using these channels, various invariants must be established. Moreover, our formal model of these channels uses high-order functions. Given the current state of the Rodin platform, this results in a high number of interactive (manual) proofs. Also, most of the proofs in the 5th and the 6th refinements are interactively discharged. The main reason for this is the lack of appropriate automatic support in the tool for reasoning about set comprehension, disjunctions, and strict subsets.

5 Conclusions

We have presented a case study in formally developing a distributed topology discovery algorithm in Event-B. Our approach to formalizing and reasoning about stable states should be applicable to other semi-reactive systems, including other routing algorithms. Our approach is novel in how it combines refinement with arguments about convergence and disjointness of events to specify liveness properties about the system eventually stabilizing and properties of the resulting stable state.

We have presented a single development of topology discovery. In actuality, we formalized several different developments, each highlighting a different aspect of the problem, making different assumptions about the environment, and establishing different properties. For example, we first considered the case where the environment is static and we developed a terminating algorithm satisfying a strong post-condition. We also considered the case where the environment is dynamic and not necessarily stabilizing. There we had the idea of augmenting the environment with some history ($DLinks$) and using this to establish interesting, although weak invariants, e.g., corresponding to our second requirement. The current development arose from different attempts to combine these developments and exploit the standard notions of convergence and deadlock-freeness as a way to express properties holding only in stable states.

Our different developments reflect not only the many facets of the problem, but also that there was a learning process involved in understanding the problem and its solution. The observation that this process is often nontrivial and requires iteration to converge on a good solution (and there may be many) is certainly not a new. But it is an observation worth repeating and such iteration fits well a development process where one alternates between specification and proving at different levels of abstraction.

References

1. Jean-Raymond Abrial. *Modeling in Event-B: System and Software Design*. Cambridge University Press, 2008. To appear.
2. Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. An open extensible tool environment for Event-B. In Z. Liu and J. He, editors, *ICFEM 2006*, volume 4260, pages 588–605. Springer, 2006.
3. Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. A mechanically proved and incremental development of IEEE 1394 tree identify protocol. *Formal Asp. Comput.*, 14(3):215–227, 2003.
4. Jean-Raymond Abrial and Stefan Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundamenta Informaticae*, XXI, 2006.
5. Ralph-Johan Back and Reino Kurki-Suonio. Decentralization of process nets with centralized control. *Distributed Computing*, 3(2):73–87, 1989.
6. Karthikeyan Bhargavan, Davor Obradovic, and Carl A. Gunter. Formal verification of standards for distance vector routing protocols. *J. ACM*, 49(4):538–576, 2002.
7. T. Clausen, G. Hansen, L. Christensen, and G. Behrmann. The Optimized Link State Routing Protocol, Evaluation through Experiments and Simulation. *IEEE Symposium on Wireless Personal Mobile Communications*, September 2001.
8. T. Clausen, P. Jacquet, A. Laouiti, et al. Optimized Link State Routing Protocol. *Request for Comments*, 3626, 2003.
9. E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.
10. Thai Son Hoang, Hironobu Kuruma, David Basin, and Jean-Raymond Abrial. Developing topology discovery in Event-B. Technical Report 611, ETH Zurich, 11/2008.
11. Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
12. J.T. Moy. *OSPF: Anatomy of an Internet Routing Protocol*. Addison-Wesley Professional, 1998.
13. J.T. Moy et al. OSPF Version 2, 1994.
14. Rfc3626: Optimized link state routing protocol (OLSR), October 2003.
15. A Udaya Shankar and Simon S Lam. A stepwise refinement heuristic for protocol construction. *ACM Transactions on Programming Languages and Systems*, 14(3):417–461, 1992.
16. Andrew Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 2002.