Developing Topology Discovery in Event-B [☆]

Thai Son Hoang[a], Hironobu Kuruma[b], David Basin[a], Jean-Raymond Abrial[a]

[a] *Department of Computer Science, ETH Zurich*
[b] *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 in developing this algorithm is specifying the problem itself. We provide a specification that 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 prove these properties by appropriately combining proofs of invariants, 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. Our entire development has been formalized and machine-checked using the Rodin tool.

Key words: Formal Methods, Routing, Refinement, Event-B, Topology Discovery

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 [2] 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 [3] 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 [26], which is the basis for protocols such as OSPF [22, 21] and OLSR [24], every router in the network must build a graph

[☆] Part of this research was carried out within the European Commission ICT project 214158 DEPLOY (Industrial deployment of system engineering methods providing high dependability and productivity) http://www.deploy-project.eu/index.html. We thank Daniel Fischer, Matthias Schmalz, and Christoph Sprenger for their comments on drafts of this paper.

representing the network topology. In this graph (also called a link-state database), 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. Roughly speaking, this means that whenever a source node sends a packet to a reachable destination, and the packet is forwarded hop-by-hop using the local routing tables, then the packet actually reaches its destination. While there has been some work on using model checkers and theorem provers to verify properties of routing protocols (see Section 5.1 for discussion of related work), there have been relatively few case studies in using formal methods to *develop* such protocols. Our work provides some insights on how this can be done.

Second, as we will see, formally developing a topology discovery protocol is surprisingly nontrivial. The complexity is both in specifying the protocol's desired properties 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 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 their status, they may no longer be connected. In other words, their link-state databases may never converge to an accurate view of the actual network topology.

To address this problem, we present a novel approach to specifying and developing algorithms whose properties depend on the environment's dynamics. In particular, we specify the system's properties in *stable* system states (cf. Section 4.3). These are, roughly speaking, states where all nodes have maximum knowledge about the environment. We prove that when certain events are *convergent* (which means they cannot take control of the system for ever, cf. Section 2.2) and deadlock free, then stable states are reached and that this suffices for the correctness of the nodes' link-state databases.

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.

Organization. In Section 2, we introduce Event-B and the Rodin tool. Afterwards, in Section 3, we describe topology discovery, within the context of link-state routing. In Section 4, we present our formal development as well as the general development strategy behind it. Finally, in Section 5, we review related work and draw conclusions.

2. Background on Event-B

Event-B is a formalism for formalizing and developing systems whose components can be modeled as discrete transition systems. It represents a further evolution of the B-method [1], which has been simplified and is now centered around the general notion of *events*, also found in Action Systems [6] and TLA [17]. We provide a brief overview here of Event-B. Full details are provided in [2].

A development in Event-B [5] is a set of formal models. The models are built from expressions in a mathematical language, which are stored in a repository. When presenting our models, we will do so in a pretty-printed form, e.g., adding keywords and following layout conventions to aid parsing. Event-B has a semantics based on transition systems and simulation between such systems, described in [2]. We will not describe in detail the semantics here and instead just describe some of the proof obligations that are important for our development.

Event-B models are organized in terms of the two basic constructs: *contexts* and *machines*. Contexts specify the static part of a model whereas machines specify the dynamic part. Contexts may contain *carrier sets*, *constants*, *axioms*, and *theorems*. Carrier sets are similar to types [5]. Axioms constrain carrier sets and constants, whereas theorems express properties derivable from axioms. The role of a context is to isolate the parameters of a formal model (carrier sets and constants) and their properties, which are intended to hold for all instances.

2.1. Machines

Machines specify behavioral properties of Event-B models. Machines may contain *variables*, *invariants*, *theorems*, *events*, and *variants*. Variables v define the state of a machine. They are constrained by invariants $I(v)$. Possible state changes are described by events.

Events. Each event is composed of a *guard* $G(t, v)$ (the conjunction of one or more predicates) and an *action* $S(t, v)$, where t are the event's *parameters*.[1] The guard states the necessary condition under which an event may occur, and the action describes how the state variables evolve when the event occurs. An event can be represented by the term

$$\textbf{any } t \textbf{ where } G(t,v) \textbf{ then } S(t,v) \textbf{ end} \quad . \tag{1}$$

We use the short form

$$\textbf{when } G(v) \textbf{ then } S(v) \textbf{ end} \tag{2}$$

when the event does not have any parameters, and we write

$$\textbf{begin } S(v) \textbf{ end} \tag{3}$$

when, in addition, the event's guard equals *true*. A dedicated event of the form (3) is used for *initialization*. Note that events may be annotated to indicate whether they

[1] When referring to variables v and parameters t, we usually allow for multiple variables and parameters, i.e., they may be "vectors". When we later write expressions like $x \; := E(t, v)$ we mean that if x contains $n > 0$ variables, then E must also be a vector of expressions, one for each of the n variables.

refine other events and with their convergence status. We will say more about this annotation later.

The action of an event is composed of one or more *assignments* of the form

$$x \;:=\; E(t, v) \tag{4}$$

$$x \;:\in\; E(t, v) \tag{5}$$

$$x \;:\mid\; Q(t, v, x')\,, \tag{6}$$

where x are some of the variables contained in v, $E(t, v)$ is an expression, and $Q(t, v, x')$ is a predicate. In (4) and (5), x must be a single variable. Assignments of the form (4) are *deterministic*, whereas the other two forms are *nondeterministic*. In (5), x is assigned an element of a set. In (6), Q is a *before-after predicate*, which relates the values x (before the action) and x' (afterwards). (6) is the most general form of assignment and nondeterministically selects an after-state x' satisfying Q and assigns it to x. There is also a side condition on the action of an event: the variables on the left-hand side of the assignments contained in the action must be disjoint. Note that the before-after predicates for (4) and (5) are as expected; namely, $x' = E(t, v)$ and $x' \in E(t, v)$, respectively.

All assignments of an action $S(v)$ occur simultaneously, which is expressed by conjoining together their before-after predicates. Assume that x is the set of variables that are modified by some assignments (i.e., the variables appearing on any assignment's left-hand side) and y are the unmodified variables (i.e., $y = v \setminus x$), the before-after predicate of the action $S(v)$ is expressed by conjoining all before-after predicates associated with each assignment and $y = y'$ (since y are unchanged). We denoted this predicate as $\boldsymbol{S}(v, v')$.

Semantics. An Event-B model formalizes a state transition system. Each state corresponds to the values of the variables v that satisfy the invariants $I(v)$, i.e., the state space is the set $\{v \mid I(v)\}$. The system's transitions correspond to the events of the Event-B model, where each event represents an atomic step that describes a system transition. Each event therefore defines a relation $R(v, v')$ between the *pre-state* v before the event and the *post-state* v' after the event. In particular, each v in R's domain satisfies the guard $G(v)$ and each v' in the R's range satisfies the before-after predicate $\boldsymbol{S}(v, v')$ given by the action. In other words, $R(v, v') = G(v) \wedge \boldsymbol{S}(v, v')$. We will later also refer to the pairs (v, v') in the relation as instances of the event. A model's transition relation is therefore the union of the transition relations associated with each of the events. The resulting transition system may be non-deterministic either because an event involves a non-deterministic action or multiple events have overlapping guards.

Obligations. Event-B defines *proof obligations*, which must be proven to show that machines have their specified properties. We describe below the proof obligation for invariant preservation. Formal definitions of all proof obligations are given in [2]. *Invariant preservation* states that invariants are maintained whenever variables change their values. Obviously, this does not hold a priori for any combination of events and invariants and therefore must be proved. For each event, we must prove that the invariants I are *re-established* after the event is carried out. More precisely, under the assumption of the invariants I and the event's guard G, we must prove that the invariants still hold in any possible state after the event's execution given by the before-after predicate $\boldsymbol{S}(t, v, v')$. The proof obligation is as follows.

$$I(v), G(v), \boldsymbol{S}(t, v, v') \;\vdash\; I(v') \qquad \textbf{(INV)}$$

Similar proof obligations are associated with a machine's initialization event. The only difference is that there is no assumption that the invariants hold. For brevity, we do not treat initialization differently from ordinary machine events. The required modifications of the associated proof obligations are straightforward. Note that in practise, by the property of conjunctivity, we can prove the preservation of each invariant separately.

2.2. Machine Refinement

Machine refinement provides a means to introduce details about the dynamic properties of a model [5]. For more details on the theory of refinement, we refer to the Action System formalism [6], which has inspired the development of Event-B. Here we sketch some central proof obligations for machine refinement.

A machine CM can refine another machine AM. We call AM the *abstract* machine and CM the *concrete* machine. The states of the abstract machine are related to the states of the concrete machine by *gluing invariants* $J(v, w)$, where v are the variables of the abstract machine and w are the variables of the concrete machine. Note that the gluing invariants $J(v, w)$ include both the local invariants of the concrete model CM (which refers only to w) and the simulation relation that should hold between the concrete and abstract domains (which refers to both v and w).

Each event ea of the abstract machine is *refined* by one or more concrete events ec. Let the abstract event ea and concrete event ec be as follows.

$$\mathsf{ea} \;\widehat{=}\; \textbf{any } t \textbf{ where } G(t, v) \textbf{ then } S(t, v) \textbf{ end} \qquad (7)$$

$$\mathsf{ec} \;\widehat{=}\; \textbf{any } u \textbf{ where } H(u, w) \textbf{ then } T(u, w) \textbf{ end} \qquad (8)$$

Somewhat simplified, we can say that ec refines ea if the guard of ec is stronger than the guard of ea (*guard strengthening*), and the gluing invariants $J(v, w)$ establish a simulation of ec by ea (*simulation*). Intuitively, the above conditions guarantee that any trace (sequence of states) of the concrete system can be simulated by the abstract system with respect to the gluing invariants $J(v, w)$. Proving guard strengthening just amounts to proving an implication. For simulation, we must prove that ec can be simulated by ea. More precisely, under the assumption of the invariants I and J and the concrete guard H, and given the transition described by $\boldsymbol{T}$, we must show that it is possible to choose a value for the abstract parameter t and a value for the abstract after variable v' such that the abstract guard G holds, the abstract before-after predicate $\boldsymbol{S}$ holds, and the gluing invariants J are re-established (this includes both the maintenance of the local invariants and preservation of the simulation relation). The proof obligation is as follows.

$$I(v), J(v, w), H(u, w), \boldsymbol{T}(u, w, w') \;\vdash\; \exists t, v' \cdot G(t) \wedge \boldsymbol{S}(t, v, v') \wedge J(v', w')$$

In order to prove the above obligation, the abstract parameter t and after variable v' need to be instantiated. The instantiations are given in the model as witnesses for t and v' associated with the concrete events. The witnesses are indicated using the keyword **with** and are given by predicates $W_1(t, u, w)$ for t and $W_2(v', u, w)$ for v'. Given the witnesses, this proof obligation can be split into the following 3 proof obligations.

$$I(v), J(v, w), H(u, w), W_1(t, u, w) \vdash G(t) \qquad \textbf{(GRD)}$$

$$I(v), J(v, w), H(u, w), \boldsymbol{T}(u, w, w'), W_1(t, u, w), W_2(v', u, w) \vdash \boldsymbol{S}(t, v, v') \quad \textbf{(SIM)}$$

$$I(v), J(v, w), H(u, w), \boldsymbol{T}(u, w, w'), W_2(v', u, w) \vdash J(v', w') \qquad \textbf{(INV\_REF)}$$

Note that in practise, we only need to give witnesses for parameters of the abstract event t that does not appear in the concrete events, and the abstract after variables v' when the abstract action modifying with these variables is non-deterministic, i.e. of the form (5) or (6). In the other cases, the witnesses can be derived.

A special case of refinement (called superposition refinement) is when v is kept in the refinement, i.e. $v \subseteq w$. This is the same as renaming the abstract variables v to v_0 and adding to $v_0 = v$ to the gluing invariants J. In particular, if the actions are deterministic for both abstract and concrete events, the simulation proof obligation **SIM** and invariant refinement proof obligation **INV_REF** hold if and only if the expressions assigned to v_0 and v are equivalent. Our reasoning in the later sections will often use this fact.

In the course of refinement, *new events* are often introduced into a model. New events must be proved to refine the implicit abstract event skip, which does nothing. Moreover, it may be proved that the new events do not collectively diverge. In other words, the new events cannot take control forever and hence one of the old events eventually occurs. To prove this, one gives a *variant* V, which maps a state w to a finite set. One then proves that each new event strictly decreases V. More precisely, let ev be a new event, where w is the state before executing ev and w' is the state after. Then for each such ev, w, and w', one proves that $V(w') \subsetneq V(w)$, under the additional assumptions of all invariants and of the guard of ev. Since the variant maps a state to a *finite* set, V induces a well-founded ordering on system states given by strict subset-inclusion of their images under V.

As explained above, we assume that the variant is a set expression. It can be more elaborate [5], but this is not relevant here. We call the new events that satisfy the above property *convergent*. Note that in some cases the convergence of some events cannot be immediately shown, but only in a later refinement. In this case, their convergence is *anticipated* and we must prove that $V(w') \subseteq V(w)$, that is, these anticipated events do not enlarge the variant. The convergent attribute of an event is denoted by keyword **status** with three possible values: *convergent*, *anticipated*, and *ordinary* (for events which are not convergent). Events are *ordinary* by default.

We have used the *Rodin Tool* [3] for our formal development. This is an industrial-strength tool for creating and analyzing Event-B models. It includes a proof-obligation generator and support for interactive and semi-automated theorem proving.

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 [13]. 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) [21, 22] and (optimized) link-state routing [10, 11].

Expressed graph-theoretically, each router corresponds to a node in the graph and there is an edge from node m to node n if m may directly (without the help of intermediate nodes) transmit data to n, i.e., m and n are *communication neighbors*. Note that this relationship is often symmetric, so the underlying graph is undirected. But it need not always be so, i.e., edges (representing links) may exist in only one direction, whereby the receiver cannot directly return messages to the sender [8]. The edges may also be weighted, where the weight may represent the physical distance between the connected nodes, or combine other relevant metrics (such as capacity, mean queuing and transmission delay, etc.). Finding optimal paths can then be reduced to computing shortest paths through the resulting graph.

In our case study, we will 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. This is done by having each node discover changes in its own local communication environment and communicating this information 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 within the context of routing, Figure 1 presents a simplified view of the main functionality of link-state routing. The algorithm consists of an infinite loop that runs on each node n. The loop's body nondeterministically chooses (represented by $\Box$) between three parts. From left-to-right, these parts are:

1. Detect and propagate changes.
2. Receive and process changes.

```
if DetectChange($m$,$n$) then
   UpdateLSDB($m$,$n$)
   UpdateSPFTree(LSDB)
   LSA ← CreateLSA($m$,$n$)
   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)
```

Figure 1: Link-state algorithm for node n (loop body)

3. Send information to neighboring nodes.

The first part describes how a node detects, processes, and propagates changes. Suppose a node n detects a change in the status of a link that joins some node m to n. The node n 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 m to n, 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 (i.e., not previously received), 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 n can, at anytime, start flooding the network by broadcasting information about its current link-state database. This can be implemented by n 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 n'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 of the algorithm, which broadcasts the LSDB thereby initiating flooding even when no changes are present. This is required for two reasons:

1. to handle the possibility that LSAs are lost during communication and
2. to handle the special case where disconnected parts of a network are reconnected.

(1) can occur if a link goes down during message transit. Figure 2 illustrates (2). Suppose that the network is disconnected into two subnetworks $S1$ and $S2$, which each undergo changes and at some later time become connected due to a link l coming up. Just flooding both subnetworks with an LSA describing l being up is not enough for the nodes in $S1$ to learn the topology of $S2$ 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 an abstract sketch in that it 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 such details are handled (using time stamps, sequence numbers,

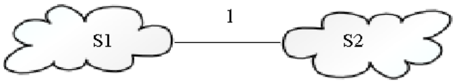


Figure 2: Link l comes up and joins two independent subnetworks

or age fields) and the updating is performed is not specified in the 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.

There is no clear agreement in the literature about the properties the protocol should have. One property sometimes mentioned is *consistency*, which is formulated in terms of actual routing decisions. Consistency states that the topology information stored by each node is such that the local routing tables that they generate lead to a loop-free path between any desired (source, destination) pair in the system. Hence data sent will not enter loops or get lost. One drawback of this specification is that it is not a property of the local states, but rather a system-wide property of routing itself. A second, more serious problem is that this property, in general, will not always hold since the local view of nodes (their LSDBs) will not always reflect the actual network topology. Hence this property is too strong: in practice, the system will often be in an inconsistent state.

We see two options for weakening consistency to something that can hold. The first option is the one usually taken by the network community and entails the use of simulation. Namely, one simulates the network under different environments and measures the rate of data throughput. The idea here is that if the environment changes slowly with respect to the system, then we expect that routing should be possible, even if not completely reliably (reliability can be handled by transport layer protocols like TCP). Simulation can be used to make statements about the network's performance, for example, throughput and delay, as a function of the environment's dynamics. It therefore also enables a quantitative comparison of protocols.

A second option, which is the one we shall pursue, is to focus on the limiting case: the behavior of the algorithm 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 k to its neighbor m when there is a path through the graph (representing the topology) from m to n.

Following this second option, we formulate our main requirement. Recall from basic graph theory that any graph can be decomposed into a collection of (maximal) strongly-connected components. Our main system requirement is then:

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 and a link is listed as down is either down or was previously down.

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.

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

We will refer to a link from m to n as either an *outward link* from m or an *inward link* to n. 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., m must periodically announce its presence to n, or, in the bidirectional case, a handshake protocol initiated by n may be used. Note, as a result, that this assumption does not require that the receiver n immediately becomes aware of changes, but only eventually.

Environment Assumption 4. A node m may send a message to a node n only when there exists a link from m to n. Moreover, the transmission occurs in a collision-free fashion.

Note that, in practice, collision-free communication may be realized in different ways. For example, using the CSMA/CD "backoff" approach in Ethernet or by choosing the time interval between two successive transmission to be larger than the propagation delay for communication along any link.

Environment Assumption 5. When a link goes down, any messages sent on it and not yet received are lost.

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

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

Here we describe our development of topology discovery in Event-B. The approach we take, which is general to system development by refinement, is to build a series of models, where each model refines the model preceding it.

4.1. Refinement Strategy

The initial models incrementally introduce our assumptions on the environment and the system, whereas the subsequent models introduce design decisions for the resulting system. Below we provide an overview of the series of models that we constructed.

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 further details about link updates, in particular a node updates information about its direct links or receives information about links from its neighbor nodes.

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 link-state information.

Refinements 6 completes the convergence proof.

In the rest of this section, we explain these models in more detail and present representative parts of our formalization. Note that the entire development (all proof obligations and theorems) have been proved using the Rodin-tool. The entire machine-checked development archive can be found on the web.[2]

[2] URL: `http://deploy-eprints.ecs.soton.ac.uk/31/`

4.2. 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*.

sets: *NODES*

constants: $closure$

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

Note that ";" denotes forward relational composition.

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 are down. These sets are disjoint (**inv0_3**) since a link cannot be simultaneously 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 $RLinks \cup DLinks$. The sets $RLinks$ and $DLinks$ are initially both empty.

In our model, we also use two auxiliary variables to track the history of the links: $RLinksH$ (H for history) represents the set of links that are up or were up. Similarly, $DLinksH$ represents the set of links that are down or were down. These are each initially assigned the empty set. The invariants **inv0_4–inv0_7** formalize the relationships between the actual links and the history links.

inv0_4–inv0_5: The history should not be too small, i.e., it should contain at least the current set of links.

inv0_6–inv0_7: The history should not be too large, i.e., it should not contain any unknown links.

The history variables $RLinksH$ and $DLinksH$ are fictional in the sense that the algorithm we develop will not actually make use of them. We will remove them from our model in a later refinement.

variables: $RLinks, DLinks, RLinksH, DLinksH$

```
invariants:
  inv0_1  $RLinks \in NODES \leftrightarrow NODES$
  inv0_2  $DLinks \in NODES \leftrightarrow NODES$
  inv0_3  $RLinks \cap DLinks = \varnothing$
  inv0_4  $RLinks \subseteq RLinksH$
  inv0_5  $DLinks \subseteq DLinksH$
  inv0_6  $RLinksH \subseteq RLinks \cup DLinks$
  inv0_7  $DLinksH \subseteq RLinks \cup DLinks$
```

```
init
  begin
    $RLinks, DLinks := \varnothing, \varnothing$
    $RLinksH, DLinksH := \varnothing, \varnothing$
  end
```

Beside initialization, there are two additional events: AddLink and RemoveLink. The first models the case where an arbitrary $link$ (that is not currently up) comes up. This link is then added to the set $RLinks$ and $RLinksH$ and removed from the set $DLinks$ (if it is already there). The second handles the symmetric case.

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

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

Note that these events formalize **Environment Assumption 2**. The fact that communication links are directed is formalized by the fact that the relations $RLinks$ and $DLinks$ are not necessarily symmetric.

4.3. 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  $\forall n \cdot rlinks(n) \subseteq RLinksH$
  inv1_4  $\forall n \cdot dlinks(n) \subseteq DLinksH$
  inv1_5  $\forall n \cdot rlinks(n) \cap dlinks(n) = \varnothing$
```

The first two invariants specify that $rlinks$ and $dlinks$ are both total functions. This formalizes that each node stores its own local information (a binary relation between *NODES*) about the status of links. Invariants **inv1_3** and **inv1_4** directly establish **System Requirement 2**: if a node has some information that a link is up, then this link must be either currently up or was up in the past, and similarly with information about down-links. 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.

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
    $\forall m, n \cdot m \mapsto n \in RLinks \Leftrightarrow m \mapsto n \in rlinks(n)$
    $\forall m, n \cdot m \mapsto n \in DLinks \Leftrightarrow m \mapsto n \in dlinks(n)$
    $\forall m, n \cdot m \mapsto n \in \mathit{closure}(RLinks) \Rightarrow$
      $(\forall k \cdot (k \mapsto m \in rlinks(n) \Leftrightarrow k \mapsto m \in rlinks(m)) \wedge$
          $(k \mapsto m \in dlinks(n) \Leftrightarrow k \mapsto m \in dlinks(m)))$
  then
    skip
  end
```

The three guards can be understood as follows.

- The first two guards hold in states where every node n knows the correct status of all its inward links. In other words, n has detected all the changes in the environment with respect to its inward links. This detection is realized in subsequent refinement levels through hello and goodbye events. Note that $m \mapsto n$ is the Event-B notation for the pair (m, n).

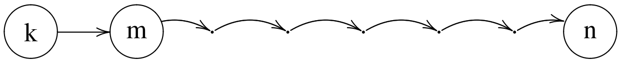


Figure 3: Information propagation from m to n

- The last guard says that if there is a path from a node m to n, i.e., $m \mapsto n \in$ *closure*$(RLinks)$, then n has the same information (up/down) as m for all inward links to m. This is illustrated in Figure 3.

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 along links. We will say that the *system is in a stable state* when the observer event can fire.

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.

$$\begin{array}{l} \quad grdStabilize \\ \Rightarrow \\ \quad (\forall M \cdot \\ \qquad (\forall f, l \cdot f \in M \land l \in M \land f \neq l \Rightarrow f \mapsto l \in closure(RLinks)) \\ \quad \Rightarrow \\ \qquad (\forall n \cdot n \in M \Rightarrow \\ \qquad\quad M \lhd rlinks(n) \rhd M = M \lhd RLinks \rhd M \quad \land \\ \qquad\quad M \lhd dlinks(n) \rhd M = M \lhd DLinks \rhd M)) \end{array}$$

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, i.e., the vertices of 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 4.9.

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
    $n \in NODES$
    $link \in RLinksH$
  then
    $rlinks(n) := rlinks(n) \cup \{link\}$
    $dlinks(n) := dlinks(n) \setminus \{link\}$
  end
```

```
removelink
  status  $anticipated$
  any  $n, link$  where
    $n \in NODES$
    $link \in DLinksH$
  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 $link$ which is currently up or was previously up. 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 $DLinks$ and only was up in the past. This reflects a key aspect of our distributed algorithm: the information nodes receive about the environment may be out-dated. But by the time n receives information that $link$ is up, the link may actually be down.

The second event removelink is analogous to addlink. From now on, we concentrate on the refinement of addlink; the refinement of removelink can be found in our on-line development archive.

Observe that since none of the three new events modifies the old variables $RLinks$ $DLinks$, $RLinksH$, and $DLinksH$, they all constitute trivial refinements of skip. At this level of refinement, addlink and removelink are *anticipated*. That is, we delay the proof that these events converge to subsequent refinements.

4.4. 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.

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

This models the situation where a node n discovers information (by receiving a "hello" message) from a node m with an outward link to n. As indicated by the **refines** keyword, this event refines the abstract event addlink, where the abstract parameter $link$ is represented by the pair $m \mapsto n$. To see that this is a refinement, observe that the guard strengthening (**GRD**) proof obligation holds since the guard of this event $m \mapsto n \in RLinks$ implies that $m \mapsto n \in RLinksH$ (recall the invariant **inv0_3**, which states that $RLinks \subseteq RLinksH$). Moreover, the proof obligations (**SIM**) and (**INV_REF**) hold since the updates of $rlinks$ and $dlinks$ are equal, with the witness $link = m \mapsto n$.

The second case models an indirect update by the transfer_rlink event.

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

This 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 pair $x \mapsto y$. The guard strengthening (**GRD**) is trivial since we did not remove the abstract guard. The proof obligations (**SIM**) and (**GRD**) are trivially satisfied with $link$ replaced by $x \mapsto y$ (witness $link = x \mapsto y$). Note that the third guard, which refers to $RLinksH$, cheats in the sense that it looks at the history. This cheating will be eliminated in a later refinement step when this event is refined and the variable $RLinksH$ is removed.

The link-state information for down-links is modeled analogously by 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. Hence, they are anticipated at this level. The reason for decomposing the convergence proof into different refinements is that this allows us to 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.

The variant that we used in this refinement is V_1 defined by

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

This is the set of inward links to n, where n has incorrect information. Since the set of *NODES* is finite, this variant is also finite. Informally, since the hello and goodbye events both provide correct information about one inward link of a node, they therefore decrease the variant V_1.

As noted above, even though we do not prove the convergence of the transfer_rlink and transfer_dlink events here, we must prove that these events do not increase the variant V_1. This is the case since these events do not change the status of any inward link to a node (notice the guard $n \neq y$), so V_1 will not be changed.

4.5. The Third Refinement

In the following 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.

In this model, we introduce a new variable, $seqNum$, representing the sequence number stored at each node for each link.

invariants:

inv3_1 $seqNum \in \mathit{NODES} \rightarrow (\mathit{NODES} \times \mathit{NODES} \rightarrow \mathbb{N})$

inv3_2 $\forall k, m, n \cdot seqNum(k)(m \mapsto n) \leq seqNum(n)(m \mapsto n)$

inv3_3 $\forall m, n, link \cdot$
$$seqNum(m)(link) = seqNum(n)(link) \land link \in rlinks(m) \Rightarrow link \in rlinks(n)$$

inv3_4 $\forall m, n, link \cdot$
$$seqNum(m)(link) = seqNum(n)(link) \land link \in dlinks(m) \Rightarrow link \in dlinks(n)$$

inv3_5 $\forall n, link \cdot 0 < seqNum(n)(link) \Rightarrow link \in rlinks(n) \cup dlinks(n)$

inv3_6 $\forall n, link \cdot link \in rlinks(n) \cup dlinks(n) \Rightarrow 0 < seqNum(n)(link)$

The events we will give preserve the following invariants:

inv3_1: Each node stores its own sequence number information about the links. This is represented as a table of non-negative numbers, with an entry for each link. The entry 0 signifies that the node does not currently have any information about the given link.

inv3_2: The sequence number n has about a link $m \mapsto n$ is always the most recent.

inv3_3–inv3_4: If two nodes m and n have the same sequence number for a given $link$, then they also have the same link-state information for that $link$.

inv3_5–inv3_6: For any node n, possessing information about a given $link$ is equivalent to having a positive sequence number for $link$.

Moreover, in order 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 events. Hence it does not affect the execution and we can therefore safely remove this variable in the subsequent refinement. The invariants concerning msg are as follows.

invariants:

inv3_7 $msg \in (\mathit{NODES} \times \mathit{NODES} \times \mathbb{N}) \leftrightarrow \mathit{NODES}$

inv3_8 $\forall x, y, sn, n \cdot$
$$\begin{aligned} & sn \leq seqNum(y)(x \mapsto y) \land \\ & seqNum(n)(x \mapsto y) < sn \\ \Rightarrow & \\ & x \mapsto y \mapsto sn \mapsto n \in msg \end{aligned}$$

inv3_9 $\text{finite}(msg)$

inv3_7: Each message contains information in the form of a link and sequence number as well as the destination node for the information.

inv3_8: If n's sequence number for a link $x \mapsto y$ is less than y's, then the information about $x \mapsto y$ from y has not yet reached n.

inv3_9: msg is finite.

In the initialization event, the sequence number for all links is set to 0 and msg is empty.

$$\begin{array}{l} seqNum := NODES \times \{(NODES \times NODES) \times \{0\}\} \\ msg := \varnothing \end{array}$$

The sequence number for a given 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
    $rlinks(n) := rlinks(n) \cup \{m \mapsto n\}$
    $dlinks(n) := dlinks(n) \setminus \{m \mapsto n\}$
    $seqNum(n)(m \mapsto n) := seqNum(n)(m \mapsto n) + 1$
    $msg := msg \cup$
            $(\{m \mapsto n \mapsto seqNum(n)(m \mapsto n) + 1\} \times (\textit{NODES} \setminus \{n\}))$
  end
```

The only difference with the abstract version is the last two assignments, which increment the sequence number and update msg.[3] 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 the link-state information being transferred is not updated, but simply passed from one node to another.

[3] The notation $f(x) := E$ denotes the update $f := f \mathbin{\lhd\!\!-} \{x \mapsto E\}$, where $\mathbin{\lhd\!\!-}$ is the operator for relational override. Note, in the third assignment, that $seqNum(n)$ is a function and therefore $seqNum(n)(m \mapsto n)$ denotes the one-point update of this function at the point $m \mapsto n$.

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

Compared to the abstract version of the event, there is an additional parameter sn. This parameter represents the sequence number that m stored for the link $x \mapsto y$ when the message was sent. This is less than or equal to the current sequence number that m has for this link, since the sequence number that a node associates with a link never decreases (it is strictly less if m has received new information on this link in the meantime). The fourth guard states that for any node k with the same sequence number for the link $x \mapsto y$, the link is in the set of k's up-links. This ensures that there will be no conflicting information in the network. Note that both the second and fourth guards (together with the last guard, introduced previously) cheat in the sense that they 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 establish guard strengthening (**GRD**) as follows. From the event's guard, we can derive that $seqNum(m)(x \mapsto y)$ is positive. Together with the invariant **inv3_5**, this implies that $x \mapsto y \in rlinks(m) \cup dlinks(m)$ (i.e. m has previously received information about the link $x \mapsto y$). We now prove $n \neq y$ by contradiction. From the second and third guards of the event, we derive that $seqNum(n)(x \mapsto y) < seqNum(m)(x \mapsto y)$ and by replacing y with n, we have $seqNum(n)(x \mapsto n) < seqNum(m)(x \mapsto n)$. However, from invariant **inv3_2**, $seqNum(m)(x \mapsto n) \leq seqNum(n)(x \mapsto n)$, which is a contradiction. The third abstract guard, i.e., $x \mapsto y \in RLinksH$, is copied here. For the proof obligations (**SIM**) and (**GRD**), the only additional assignments are to update the sequence number and msg. Hence this obligation is trivially satisfied.

In this refinement, we also proved the convergence of the transfer_rlink and transfer_dlink events. The variant V_2 is just msg. First, by **inv3_9**, the variant is finite. Second, the action of these two *transfer* events remove $x \mapsto y \mapsto sn \mapsto n$ from msg. Finally, from the invariant **inv3_8** and the guard of this event, $x \mapsto y \mapsto sn \mapsto n \in msg$. Hence these events decrease the variant V_2.

The variants V_1 and V_2 form a lexicographical variant, namely $V = (V_2, V_1)$ where V_2 has higher precedence. The convergence proofs that we gave in the current and the last refinement show that the events hello, goodbye, transfer_rlink, and transfer_dlink decrease the combined variant V.

The guard of the observer event stabilize is also refined using information about sequence numbers. In particular, the abstract event

```
stabilize
  when
    $\forall m, n \cdot m \mapsto n \in RLinks \Leftrightarrow m \mapsto n \in rlinks(n)$
    $\forall m, n \cdot m \mapsto n \in DLinks \Leftrightarrow m \mapsto n \in dlinks(n)$
    $\forall m, n \cdot m \mapsto n \in \mathit{closure}(RLinks) \Rightarrow$
      $(\forall k \cdot (k \mapsto m \in rlinks(n) \Leftrightarrow k \mapsto m \in rlinks(m)) \wedge$
        $(k \mapsto m \in dlinks(n) \Leftrightarrow k \mapsto m \in dlinks(m)))$
  then
    skip
  end
```

becomes

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

The first two guards are unchanged and state that every node knows the status of all inward links. What is new is the last guard. This states that for any pair of nodes m and n, and link $link$, if m has a direct communication link to n, then n's information about $link$ is not older than m's. From the properties of *closure* and invariant **inv3_2**, it follows that if there is a path from m to n, then n will have the same sequence number for all links inward to m. This fact, together with the invariants **inv3_3** and **inv3_4**, allows us to conclude that n will have up-to-date information about all inward links to m (which is the last abstract guard).

4.6. 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: $SChan$, $RChan$, and $DChan$. These model the channels for transmitting sequence numbers, up-link information, and down-link information, respectively.

invariants:

inv4_1 $SChan \in (\mathit{NODES} \times \mathit{NODES}) \rightarrow ((\mathit{NODES} \times \mathit{NODES}) \rightarrow \mathbb{N})$

inv4_2 $RChan \in (\mathit{NODES} \times \mathit{NODES}) \rightarrow (\mathit{NODES} \leftrightarrow \mathit{NODES})$

inv4_3 $DChan \in (\mathit{NODES} \times \mathit{NODES}) \rightarrow (\mathit{NODES} \leftrightarrow \mathit{NODES})$

For each pair of nodes, the link-state (up/down) information is a relation between *NODES*, formalizing the set of pairs of nodes on the communication channel. More precisely, 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.

We now mention the relevant channel properties.

invariants:

inv4_4 $\forall m, n \cdot RChan(m \mapsto n) \cap DChan(m \mapsto n) = \varnothing$

inv4_5 $\forall m, n \cdot (\exists link \cdot 0 < SChan(m \mapsto n)(link)) \Rightarrow m \mapsto n \in RLinks$

inv4_6 $\forall m, n, link \cdot SChan(m \mapsto n)(link) \leq seqNum(m)(link)$

inv4_4: Link-state channels from nodes m to n are disjoint.

inv4_5: If there is traffic (i.e., a $link$ with a positive sequence number) in the channel from m to n, then the link $m \mapsto n$ must currently be up.

inv4_6: For any two nodes m and n and a $link$, $link$'s sequence number in the channel from m to n is not newer than the sequence number stored at node m for the same link.

invariants:

inv4_7 $\forall m, n, link \cdot link \in RChan(m \mapsto n) \Rightarrow$
$(\forall k \cdot seqNum(k)(link) = SChan(m \mapsto n)(link) \Rightarrow$
$link \in rlinks(k))$

inv4_8 $\forall m, n, link \cdot link \in DChan(m \mapsto n) \Rightarrow$
$(\forall k \cdot seqNum(k)(link) = SChan(m \mapsto n)(link) \Rightarrow$
$link \in dlinks(k))$

inv4_9 $\forall k, link \cdot link \in rlinks(k) \Rightarrow$
$(\forall m, n \cdot seqNum(k)(link) = SChan(m \mapsto n)(link)$
$\Rightarrow link \in RChan(m \mapsto n))$

inv4_7 – inv4_9: The sequence numbers in the channels are consistent with the sequence numbers stored at each node. For example, **inv4_7** states that if a $link$ is in the channel for up-links from m to n, then for any node k which has the same sequence number as that stored in channel from m to n, $link$ must be in the set of up-links of the node k.

Note that the statement corresponding to **inv4_9** for down-links, i.e.

$$\begin{array}{l}\forall k, link \cdot link \in dlinks(k) \Rightarrow \\ \quad (\forall m, n \cdot seqNum(k)(link) = SChan(m \mapsto n)(link) \\ \qquad \Rightarrow link \in DChan(m \mapsto n))\ ,\end{array}$$

is derivable from the set of invariants.

invariants:

inv4_10 $\forall m, n, x, y, link \cdot$
$SChan(m \mapsto n)(link) = SChan(x \mapsto y)(link) \land$
$link \in RChan(m \mapsto n)$
$\Rightarrow$
$link \in RChan(x \mapsto y)$

inv4_11 $\forall m, n, link \cdot link \in RChan(m \mapsto n) \Rightarrow$
$0 < SChan(m \mapsto n)(link)$

inv4_12 $\forall m, n, link \cdot link \in DChan(m \mapsto n) \Rightarrow$
$0 < SChan(m \mapsto n)(link)$

inv4_13 $\forall m, n, link \cdot link \notin RChan(m \mapsto n) \land link \notin DChan(m \mapsto n)$
$\Rightarrow SChan(m \mapsto n)(link) = 0$

inv4_10: The sequence numbers in the channels are consistent with each other. For example, if a $link$ has the same sequence number in the channel from m to n and the channel from x to y, then this $link$ either belongs to the up channels of both $m \mapsto n$ and $x \mapsto y$, or the down channels of both, but not up for one and down for the other.

inv4_11 – inv4_13: For each pair of nodes m and n and the link $link$, if $link$ is in one of the link-state channels, then the sequence number for $link$ in $SChan$ is also positive and vice versa.

Moreover, at this stage, we can remove the history variable $RLinksH$ and $DLinksH$. To prove refinement, we need the following invariants, which relate these history variables to the information in the channels.

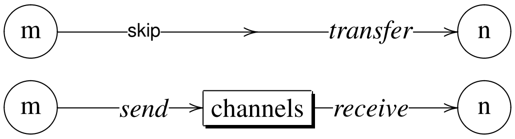
invariants:

inv4_14 $\forall m, n \cdot RChan(m \mapsto n) \subseteq RLinksH$

inv4_15 $\forall m, n \cdot DChan(m \mapsto n) \subseteq DLinksH$

inv4_14 – inv4_15: For each pair of nodes m and n, the up-link information in the channel from m to n is included in $RLinksH$, the set of links that are up or were up. The invariant for down-links is analogous.

Coming back to the modeling of the events, the actual 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 (*send* and *receive*).



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
    $SChan(m \mapsto n)(link) seqNum(m)(link)$
    $RChan(m \mapsto n) := RChan(m \mapsto n) \cup \{link\}$
  end
```

For a node to send information about a $link$, this event assumes that the information about the same $link$ from the last send has been received or lost, see **Environment Assumption 4**. 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), which is also required by **Environment Assumption 4**.

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
    $rlinks(n) := rlinks(n) \cup \{x \mapsto y\}$
    $dlinks(n) := dlinks(n) \setminus \{x \mapsto y\}$
    $seqNum(n)(m \mapsto n) := SChan(m \mapsto n)(x \mapsto y)$
    $SChan(m \mapsto n)(x \mapsto y) := 0$
    $RChan(m \mapsto n) := RChan(m \mapsto n) \setminus \{x \mapsto y\}$
  end
```

The link-state information is retrieved from the channels from m to n. Here, the abstract parameter sn is refined as $SChan(m \mapsto n)(x \mapsto y)$. Note that the proof obligations (**SIM**) and (**GRD**) are trivially satisfied since the additional actions only modify new variables, namely $SChan$ and $RChan$. To establish guard strengthening (**GRD**), we must prove the following.

- $m \mapsto n$ is an up-link. But, since $seqNum(n)(x \mapsto y) < SChan(m \mapsto n)(x \mapsto y)$, we know that $SChan(m \mapsto n)(x \mapsto y)$ is positive. From the invariant **inv4_5**, we can conclude that the link $m \mapsto n$ is an up link.

- $SChan(m \mapsto n)(x \mapsto y)$ (as a refinement of the abstract parameter sn) satisfies the guard of the abstract event, i.e.

  $$SChan(m \mapsto n)(x \mapsto y) \leq seqNum(m)(x \mapsto y)$$
  $$seqNum(n)(x \mapsto y) < SChan(m \mapsto n)(x \mapsto y)$$
  $$\forall k \cdot seqNum(k)(x \mapsto y) = SChan(m \mapsto n)(x \mapsto y) \Rightarrow x \mapsto y \in rlinks(k)$$

  The first condition follows from the invariant **inv4_6**. The second condition is exactly the first guard of this concrete event. The last condition can be derived from the second guard, $x \mapsto y \in RChan(m \mapsto n)$, and the invariant **inv4_7**.

- $x \mapsto y \in RLinksH$. But we know that $x \mapsto y \in RChan(m \mapsto n)$ and from invariant **inv4_14**, we have that $RChan(m \mapsto n) \subseteq RLinksH$ and hence $x \mapsto y \in RLinksH$.

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-links and down-links. Another reason for introducing discard events is that, without them, we would not be able to prove deadlock freedom 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
    $SChan(m \mapsto n)(link) := 0$
    $RChan(m \mapsto n) := RChan(m \mapsto n) \setminus \{link\}$
  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 5**. 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
    $RLinks := RLinks \setminus \{link\}$
    $DLinks := DLinks \cup \{link\}$
    $SChan := SChan \mathbin{\lhd\!\!\!\!-} (\{link\} \times \{NODES \times NODES \times \{0\}\})$
    $RChan(link) := \varnothing$
    $DChan(link) := \varnothing$
  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.

4.7. The Fifth Refinement

Our machine in the fourth refinement is an implementation of the protocol. However, we have not yet established the convergence of the events send_rlink and discard_rlink (and correspondingly for $dlink$). We are now faced with the following problem: these events actually do not converge and should not converge. As we saw in Figure 1 (third part), each node will periodically broadcast information about its LSDB and its neighbors will repeatedly receive this information, even when it is not new. What we will show then is that the system eventually does reach a stable state (assuming that the environment does not change), i.e. the system satisfies **System Requirement 1**, despite continually broadcasting and receiving redundant information.

To prove this, we construct an equivalent model of the system by first partitioning 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. This is done in this section and Section 4.8. Afterwards, in Section 4.9, we prove that stability follows from this partial convergence, under an additional assumption concerning the *strong-fairness* of event execution.

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 be propagated to n and therefore further information coming from m about $link$ will not change this neighbor-stable status. Using this notion of neighbor stable, we can restate the third guard of the observe event stabilize (from Section 4.5) 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

$$\boxed{seqNum(m)(link) \leq seqNum(n)(link)}$$

or its complement. For example, we partition the send_rlink event 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.

To prove that the events send_rlink_unstable and send_dlink_unstable are convergent, we use the following variant V_3.

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

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.

The convergence proof is as follows. First, note that all these events transfer $link$'s sequence number from m to n. For any tuple $x \mapsto y \mapsto k$ different from $m \mapsto n \mapsto link$, the events neither change $SChan(x \mapsto y)(k)$ nor $seqNum(y)(k)$. Hence, we can restrict our attention to $m \mapsto n \mapsto link$. Now consider the following cases.

- For the events send_rlink_unstable and send_dlink_unstable, their guards state that the sequence number for $link$ in the channel from m to n is 0 and hence $m \mapsto n \mapsto link \in V_3$. After the event, the sequence number for $link$ in m, which is newer than n's sequence number for $link$, is copied to the channel. Hence $m \mapsto n \mapsto link \notin V_3'$ (V_3' denotes the value of the variant after the event execution) and therefore V_3 is decreased.

- For the events send_rlink_stable and send_dlink_stable, their guards state that the sequence number in the channel is 0. Hence $m \mapsto n \mapsto link \in V_3$. After the event, the information from m that is not newer than that of n is copied to the channel. Hence $m \mapsto n \mapsto link \in V_3'$. This means that V_3 does not increase.

- For discard_rlink_stable, discard_rlink_unstable, discard_dlink_stable, and discard_dlink_unstable, the guards of these events state that the information in the channel before is not newer than that of n and afterwards this information is reset to 0, which is also not newer than that n. Hence V_3 also does not increase.

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 4 events send_rlink_stable, send_dlink_stable, discard_rlink_stable, and discard_dlink_stable maintain the system's stable state, that is, if the state before the event execution is stable then the state after the event execution is also stable. This is easy to prove since $stable$ refers only to $RLinks$, $DLinks$, $rlinks$, $dlinks$, and $seqNum$, whereas our 4 events only modify the information in the channels, i.e., $SChan$, $RChan$, and $DChan$. Hence, these events will maintain the stable state.

4.8. Sixth Refinement

In this refinement step, we prove the convergence of the discard_rlink_unstable and discard_dlink_unstable. The variant V_4 that we used is

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

Informally, the variant represents the set of messages about $link$ that are transferred from m to n, where $link$ is not neighbor stable from m to n. The proof is as follows.

- The events discard_rlink_unstable and discard_dlink_unstable discard a message for a $link$ from m to n where the information is unstable. Hence they decrease the variant V_4.

- The events discard_rlink_stable and discard_dlink_stable also discard a message for a $link$ from m to n, but the information is stable. Hence they do not increase the variant V_4.

4.9. 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, this is sufficient 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 agrees 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 level, the set of events can be divided into different groups as follows.

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,

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

where $G(CE_i)$ is the guard of the event C_i.

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, …, Env_l from the first group occur) then the system will eventually reach a stable state and remain in this state.

The following proof is a traditional "paper and pencil proof", rather than a proof using the Rodin tool.

PROOF. 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. □

Note that the theorem statement is closely related to the proof rules for extended response of Manna and Pnueli [19]. Our statement is somewhat simpler than their rules as we deal only with assertional (state) formulas and strongly fair events (they consider both weakly and strongly fair transitions). Moreover, we have an additional assumption (iv), which we use to establish that stability is preserved after a stable state is reached.

In our application of this theorem, we assume Proposition (v), whereas the other propositions have already been previously proved using the Rodin tool. In particular, we proved Propositions (i) and (iv) in the 5th refinement and Propositions (ii) and (iii) in the 2nd, 3rd, 5th, and 6th refinements. 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 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 (recall Section 2.1). 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 4.3, 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**.

4.10. 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 guarantee 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 large 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 automatic support in the tool for reasoning about set comprehension, disjunctions, and strict subsets.

Model	Number of Proof Obligations	Automatically Discharged	Interactively Discharged
Context	0	0	0
Initial Model	21	19(91%)	2(9%)
1st Refinement	33	30(91%)	3(9%)
2nd Refinement	30	25(83%)	5(17%)
3rd Refinement	74	38(54%)	36(46%
4th Refinement	176	102(58%)	74(42%)
5th Refinement	44	7(16%)	37(84%)
6th Refinement	8	0(0%)	8(100%)
Total	386	221(57%)	165(43%)

Table 1: Proof statistics

5. Related Work and Conclusions

5.1. Related Work

Numerous formal methods have been applied to the analysis of network protocols. This includes model checking [7, 16], theorem proving [12], and development by refinement [4, 25]. Most of the existing case studies focus on *endpoint* protocols, such as link-layer protocols like the sliding-window or alternating-bit protocols, or higher-level protocols such as SSL/TLS. These protocols generally involve just two processes (the end points) or perhaps a third process (e.g., an adversary). Routing is different as its specification should make a general statement about an entire networks of nodes, executing the protocol concurrently.

With respect to routing protocols, probably the most detailed study is that of [9], who used an interactive theorem prover (HOL) together with a model checker (SPIN) to prove different properties of distance vector routing protocols. They carried out case studies analyzing the Routing Information Protocol (RIP) standard and the Ad-Hoc On-Demand Distance Vector (AODV) protocol. Although the protocols they analyze are of a different flavor than ours (distance vector versus link state) there are a number of similarities. For instance, in their analysis of RIP, they formalize a notion of *stability*, which captures nodes agreeing on shortest paths. They are able to establish this property in general, since the protocol imposes limits on the lengths of paths (so-called hop counts). In contrast, we can only show (our notion of) the stability of topology discovery under the assumption of a suitably quiescent environment. Another substantial difference is they carry out post-hoc protocol verification whereas we focus on protocol development.

In [23], the authors describes their use of CMC, a code-based model-checker for C and C++, to model check different implementations of AODV. They use model checking not for verification, but rather for bug finding and hence they can soundly reduce the protocol's infinite state space (unbounded number of nodes, unbounded sequence counters, etc.) to a finite one by scaling down their model to work with a fixed number (2 to 4) of processes that operate on data from finite domains. The properties checked include properties of the distributed routing tables (which was also the case in [9]), such as the routing tables of all nodes do not form a loop. In addition, since they are working with a code-based checker, they are able to search for implementation errors, such as segmentation violations, memory leaks, dangling pointers, and the like. These implementation aspects are, of course, not present in our work, although it is possible in theory to carry out the refinement down to actual code, which is then, by construction, error free. The Rodin tool does not yet, however, support this.

A number of network protocols have been formally developed using refinement. For example, [25] shows how to develop a family of different sliding window protocols. These are two party endpoint protocols that provide reliable data transfer between a producer and a consumer connected by unreliable channels. An example of non-endpoint protocol is given in [4], who present the development of a distributed leader election protocol on a connected network graph (the IEEE 1394 protocol). [2] presents the development in Event-B of a routing algorithm for mobile agents due to [20], which was originally verified in Coq.

Finally, note that the main system property that we show (**System Requirement 1**) is established by proving that the system enters a stable state. The notion of stability that we formalize in Section 4.3 is an instance of the general notion of a *stable system property* (see e.g., [14, 18]), 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. Different approaches have been given for proving stabilization properties of protocols, e.g., [15, 27]. Our Theorem 2 gives sufficient conditions for establishing (a form of) stability. It is attractive in that, with the exception of the fairness assumption, all other assumptions can directly and easily be established with the Rodin tool.

5.2. 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 particularly 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. However, 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 history variables and using them to establish interesting, although weak invariants, e.g., corresponding to our second requirement. The current development, and our general development approach, 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 the fact that there was a learning process involved in understanding the problem, the solution, and the invariants that hold. The observation that specifying problems 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. *The B-book: assigning programs to meanings*. Cambridge University Press, 1996.

[2] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2009. To appear.

[3] 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.

[4] 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.

[5] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundamenta Informaticae*, XXI, 2006.

[6] Ralph-Johan Back and Reino Kurki-Suonio. Decentralization of process nets with centralized control. *Distributed Computing*, 3(2):73–87, 1989.

[7] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[8] Lichun Bao and J.J. Garcia-Luna-Aceves. Link-state routing in networks with unidirectional links. In *In Proceedings Eight International Conference on Computer Communications and Networks*, pages 358–363, 1999.

[9] Karthikeyan Bhargavan, Davor Obradovic, and Carl A. Gunter. Formal verification of standards for distance vector routing protocols. *J. ACM*, 49(4):538–576, 2002.

[10] 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.

[11] T. Clausen, P. Jacquet, A. Laouiti, et al. Optimized Link State Routing Protocol. *Request for Comments*, 3626, 2003.

[12] Marco Devillers, David Griffioen, Judi Romijn, and Frits Vaandrager. Verification of a leader election protocol: Formal methods applied to ieee 1394. *Form. Methods Syst. Des.*, 16(3):307–320, 2000.

[13] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[14] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[15] Mohamed G. Gouda and Nicholas J. Multari. Stabilizing communication protocols. *IEEE Trans. Comput.*, 40(4):448–458, 1991.

[16] Gerhard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison–Wesley, 2003.

[17] Leslie Lamport. The temporal logic of actions. *Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, May 1994.

[18] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[19] Zohar Manna and Amir Pnueli. Completing the temporal picture. *Theoretical Computer Science*, 83(1):97–139, 1991.

[20] Luc Moreau. Distributed directory service and message routing for mobile agents. *Sci. Comput. Program.*, 39(2-3):249–272, 2001.

[21] J.T. Moy. *OSPF: Anatomy of an Internet Routing Protocol*. Addison-Wesley Professional, 1998.

[22] J.T. Moy et al. OSPF Version 2, 1994.

[23] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. Cmc: a pragmatic approach to model checking real code. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 75–88, New York, NY, USA, 2002. ACM.

[24] Rfc3626: Optimized link state routing protocol (OLSR), October 2003.

[25] 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.

[26] Andrew Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 2002.

[27] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, New York, NY, USA, 2001.