Check for updates

Research Article

International Journal of Distributed Sensor Networks

International Journal of Distributed Sensor Networks
2021, Vol. 17(12)
© The Author(s) 2021
DOI: 10.1177/15501477211058667
journals.sagepub.com/home/dsn
SAGE

Modeling and proving dynamic behaviors of a routing protocol: A tutorial

Agnieszka Paszkowska and Konrad Iwanicki

Abstract

With the increasing adoption of Internet of Things technologies for controlling physical processes, their dependability becomes important. One of the fundamental functionalities on which such technologies rely for transferring information between devices is packet routing. However, while the performance of Internet of Things–oriented routing protocols has been widely studied experimentally, little work has been done on provable guarantees on their correctness in various scenarios. To stimulate this type of work, in this article, we give a tutorial on how such guarantees can be derived formally. Our focus is the dynamic behavior of distance-vector route maintenance in an evolving network. As a running example of a routing protocol, we employ routing protocol for low-power and lossy networks, and as the underlying formalism, a variant of linear temporal logic. By building a dedicated model of the protocol, we illustrate common problems, such as keeping complexity in control, modeling processing and communication, abstracting algorithms comprising the protocol, and dealing with open issues and external dependencies. Using the model to derive various safety and liveness guarantees for the protocol and conditions under which they hold, we demonstrate in turn a few proof techniques and the iterative nature of protocol verification, which facilitates obtaining results that are realistic and relevant in practice.

Keywords

Routing, protocol, routing protocol, dependability, correctness, modeling, verification, protocol verification, safety, liveness, linear temporal logic, low-power wireless network, dynamic network, RPL, IoT, industrial IoT

Date received: 19 August 2020; accepted: 15 October 2021

Handling Editor: Peio Lopez Iturri

Introduction

The goal of routing is finding paths in a network along which data packets can be sent to enable communication between nodes that are not connected directly. Routing protocols are thus fundamental in the Internet and will likely remain important in the emerging Internet of Things (IoT),[1] which aims to make physical objects part of the global network. One of the reasons is that networking such objects often requires technologies for low-power wireless communication, which has limited range and can be hindered by environmental obstacles. Consequently, forwarding packets through intermediate nodes, selected by a routing protocol, may be the only way to ensure that related physical objects are indeed capable of exchanging data.

However, designing and implementing a routing protocol is far from trivial. Apart from the various trade-offs in algorithms for selecting routing paths, a major challenge is that the topology of a network is

Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Warszawa, Poland

Corresponding author:
Konrad Iwanicki, Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, ul. Banacha 2, 02-097 Warszawa, Poland.
Email: iwanicki@mimuw.edu.pl

Creative Commons CC BY: This article is distributed under the terms of the Creative Commons Attribution 4.0 License (https://creativecommons.org/licenses/by/4.0/) which permits any use, reproduction and distribution of the work without further permission provided the original work is attributed as specified on the SAGE and Open Access pages (https://us.sagepub.com/en-us/nam/open-access-at-sage).

typically highly dynamic, that is, the node population and link qualities constantly evolve. This is especially apparent in low-power wireless networks, when nodes are embedded in the surrounding environment. In effect, a crucial element of a routing protocol is algorithms that detect such changes and account for them by adapting, rebuilding, or even tearing down completely the routing paths between nodes. The operation of such route maintenance algorithms is inherently distributed among the nodes and can be influenced by external (environmental) factors. In addition, even standardized algorithms usually have a number of configuration parameters, rely on external components (e.g. for detecting failures), or even leave some issues open to implementers.

As a result, it may be difficult to predict for a routing protocol how its given implementation in a specific configuration and particular network will behave in a given scenario of network topology dynamics. Although such behavior can sometimes be tested empirically, some scenarios, even ones that are likely in the real world, may be difficult to reproduce during pre-deployment testing. What is more, even if tests in given conditions are possible, they provide only a limited understanding of how the conditions are allowed to change and what hazards such changes entail. This implies that deploying a routing protocol in a real-world system poses some risks. While often these risks are simply ignored, there are use cases for which they have to be given more consideration. A prominent example is many industrial IoT applications, especially involving actuation of valuable assets. Such applications frequently require a high degree of dependability,[2] including guarantees on the behavior of the employed routing protocol under various network dynamics.

In this article, we give a tutorial on how such guarantees can be derived formally. As the considered routing protocol we select the IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL).[3] RPL is the current IETF standard for routing IPv6 packets in low-power wireless networks, developed to allow such networks to be part of IoT. It has a couple of implementations, among which two open-source ones, ContikiRPL[4] and TinyRPL,[5] are arguably the most widely recognized and deployed for both research and commercial purposes. It also exhibits virtually all properties characteristic of such standardized solutions: its specification is rather voluminous, it relies on other IETF standards and external components, and introduces over a dozen of configurable parameters. As such, it serves well as a running example of a relevant, practical routing protocol.

The tutorial is actually inspired by real-world problems that we have encountered when deploying RPL and is based on our previous work on those problems, involving mainly modeling and verification[6,7] but also some empirical approaches.[8,9] Its goal as a whole is allowing the readers to apply a similar reasoning to produce complete proofs or counterexamples for their own hypotheses about the dynamic operation of distance-vector routing protocols, such as RPL, potentially in custom parameter configurations and deployment scenarios, to improve the dependability of those protocols and their implementations. The presented techniques can in addition help the readers who are familiar with (semi-)automated verification tools to have their results, obtained for specific settings, generalized to a range of possible deployments. Overall, as we discuss in more detail in the next section, the covered material can be of value at all stages of a protocol lifetime: it provides unique insight into an operation of a routing protocol under network topology dynamics, which can help improve its design, develop correct implementations, and configure them for particular deployment scenarios.

The tutorial targets a wide audience from the communications community. As such, it does not assume any prior experience with formal verification methods. It requires only undergraduate-level knowledge of computer logic and some background in networking. For the same reason, while it does present examples of a formal notation, that is, linear temporal logic (LTL),[10,11] it strives to explain as much as possible in a textual form, thereby following a successful approach of analyzing and proving classic distributed algorithms.[12] All in all, we hope that the tutorial is accessible not only to the scientific community but also to practitioners involved in building dependable systems.

The contributions of the tutorial are as follows:

- We start by surveying a broad spectrum of approaches to ensuring correctness of protocol implementations. This aims to position our work and provide the readers with possible alternatives.
- We then give the necessary background, that is, an overview of RPL and LTL. These sections are meant to be self-contained but they also offer relevant pointers in the case the readers needs more information.
- What follows is the first part of the tutorial core, in which we explain how selected aspects of RPL can be modeled. The section emphasizes typical problems encountered in the process: keeping complexity in control, modeling processing and communication, abstracting algorithms comprising the protocol, and dealing with open issues and external dependencies.
- In the second part of the tutorial core, we show in turn how various hypotheses regarding dynamic behaviors of the developed model can be verified. We consider hypotheses regarding

both safety and liveness, demonstrate a few proof techniques, including custom ones, and illustrate the iterative nature of a protocol verification process, which are meant to help obtaining results that are relevant in practice.
- Finally, we summarize and give possible further directions. In particular, we briefly recapitulate real-world applications of the results we have developed using the methodology presented in this tutorial.

The many facets of dependability

Dependability encompasses multiple aspects.[13] In this tutorial, we are concerned with ensuring that one can rely on implementations of a routing protocol to correctly handle network topology changes that are observable in practice. We will not try to specify here precise correctness requirements because they may vary depending not only on the protocol but also its target environment. However, the goal is to have guarantees on the behavior of a protocol implementation under as broad a spectrum of deployment settings and operational scenarios as possible, so that one is able to assess the risks and consequences of protocol malfunctions, and alleviate them, for instance, through additional dedicated software solutions or hardware overprovisioning.

This formulation of the goal has two facets. The first is ensuring that a protocol specification, that is, the algorithms constituting the protocol, including their assumptions, is correct. The second is ensuring that implementations of the algorithms conform to their specifications, that is, that they do not have bugs. The difference between these two is subtle and they are often treated together, especially since the development of a protocol is typically an iterative process alternating between specification, implementation, and testing. Therefore, we also treat them together, thereby surveying related work from a perspective of the entire protocol development cycle.

Methods for ensuring dependability

To start with, testing is crucial for assessing the performance of a routing protocol implementation in practice. It may also reveal bugs in the implementation or even in the design of the protocol itself. In low-power wireless networks, a particularly popular form of testing is integration testing, which involves entire protocol implementations or their major components. It is typically performed in simulators, such as TOSSIM[14] and OMNeT++ ,[15] emulators, like COOJA[16] and Avrora,[17] and on testbeds, for example, MoteLab,[18] Indriya,[19] FIT/IoT-LAB,[20] or 1KT.[21] Nevertheless, while integration testing is indispensable for general performance assessment, it is hardly ever capable of exercising all possible control flow paths, which is necessary for reliability. To this end, one may additionally employ finer-grained forms of testing, notably unit testing.[22,23] However, those are aimed at individual software modules and hence may be incapable of identifying bugs resulting from module interactions. Moreover, they require precise specifications of the behavior of the modules, which need not be trivial to derive from a specification for an entire protocol, especially since ideally the specification should not enforce a particular modularization. Consequently, testing alone may be insufficient to ensure that an implementation of a routing protocol is reliable, in particular, that it correctly handles network topology changes that are observable in practice.

On the contrary, appropriate solutions have to be adopted also during protocol implementation because without a sufficient quality of its code, it is difficult, if not impossible, to make an implementation reliable. In fact, unit testing can already be one example, as it is typically done together with programming.[23,24] Another popular solution is to employ modern programming languages[25] or domain-specific ones,[26] which aim to simplify software engineering and prevent certain types of bugs. Moreover, such languages are often accompanied by dedicated design pattern,[27] integrated development environments,[28] and debuggers,[29–31] the goal of which is to further improve software quality. Finally, additional compile- and run-time solutions can also be deployed to improve memory safety,[32] cross-interface behavior,[33] assertion checking,[34] (distributed) checkpointing,[35] or software updates,[36] to name just a few examples. Nevertheless, development-oriented solutions are yet unable to protect against all classes of bugs. Furthermore, they rely on programmers to correctly interpret specifications not only to produce compliant code but also to devise appropriate tests, assertions, and the like. Above all, the specifications and designs must themselves be correct; otherwise, even their highest quality implementations are bound to behave in an undesired manner.

For this reason, a protocol design process should also promote quality. One successful approach is to have it led by an expert group, supported by a broad community. This allows for leveraging multiple skilled people to identify and eliminate potential design flaws and to propose various extensions or improvements. For instance, RPL was devised by IETF's dedicated group, ROLL,[37] that engaged the low-power wireless networking community worldwide by publishing multiple working drafts and request for comments (RFCs). Likewise, its two popular implementations, ContikiRPL[4] and TinyRPL,[5] were developed in an open-source model. In this context, it is also crucial that the entire protocol development process is

iterative, which enables fixing design flaws and implementation bugs identified during testing, thereby gradually reducing their numbers. However, as online bug reports and our examples from the previous section suggest, even such an advanced process may be insufficient to produce routing protocol implementations that are suitable for highly dependable systems.

Formal verification and model checking

This is where formal methods may be of use. Depending on their type, they can be applied at all development stages of a protocol and can yield provable guarantees that the protocol, as given by a specification, or its implementation, behaves in a certain manner under certain assumptions.

A particularly appealing approach is to employ automated verification tools,[38] which for a model or actual code of (a fragment of) a concurrent program—in our case, a routing protocol—enumerate and explore all or selected execution paths in order to find states violating user-defined conditions or conclude that such states do not exist. Examples of tools for generic concurrent programs include SPIN,[39] which defines a new program modeling language, Promela, and performs automated state space exploration of programs in this language to prove user-provided time-insensitive properties or find counterexamples, UPPAAL,[40] which features a graphical modeling interface and enables verifying real-time properties, PRISM,[41] which in addition allows for a faster, probabilistic state space exploration, or MaceMC,[42] which operates on actual implementation code rather than code written in a special modeling language. Examples of tools designed specifically for low-power wireless network protocols are in turn KleeNet,[43] which checks global assertions in protocol implementations by adopting symbolic execution to explore points at which control flow in their code changes or branches, T-Check,[44] which, drawing from MaceMC,[42] uses random walks over the state space of a protocol implementation to probabilistically verify user-supplied global properties, and Anquiro,[45] which instead of probabilistic exploration introduces three levels of abstractions suitable for assessing protocols at different networking layers.

Although automated software verification did help identify bugs in various systems and protocols, including ones for low-power wireless networks,[43–45] this approach has inherent limitations. Since it requires enumerating all visited protocol states, it suffers from state space explosion. In principle, a state space grows exponentially with the number of nodes, links, local variables of the nodes and fields of messages in transit, and the different possible values they can attain. This means that despite various optimizations employed by the aforementioned tools, enumerating all relevant states is frequently infeasible. In contrast, analyzing only a subset of the states, as in probabilistic solutions, typically does not give *guarantees* that a property always holds if a verifier fails to find a counterexample. Therefore, by and large, automated verification is typically performed only for small systems, consisting of few nodes and links. What is more, any property verified in such a system is guaranteed only for this exact system. In other words, generalizing the property to other networks or different parameter settings necessitates other means.

Consequently, while automated verification is indispensable for identifying some classes of problems, many important properties are proved for protocols in a traditional way: by a human applying reasoning rules to analyze a property in a range of network topologies and configurations. Examples for low-power wireless networks include deriving conditions for node ranking that were later adopted by RPL,[46] devising algorithms for route repair with a guaranteed approximation factor,[47] finding bounds for various next-hop selection algorithms,[48] and the original research underlying this tutorial.[6,7] An important benefit of this approach is that it usually also gives deep understanding of *why* and when (under what assumptions) a given property holds.

The traditional approach, however, requires techniques that make deriving proofs doable in reasonable time and facilitate checking them. This tutorial is partially due to the fact that we have lacked such techniques that would be aimed specifically at dynamic behaviors of routing protocols. Prior approaches to proving properties of such protocols, including the aforementioned examples, typically consider a snapshot of a system, often in some stable state, and involve showing the properties for this snapshot. Our focus is in contrast dynamic behavior due to continuous changes in a network, which precludes considering just a single snapshot. This bears some similarities to the problems that the distributed systems community faces when proving claims for eventual consistency:[49] many proofs assume quiescent systems but such systems are in practice never quiescent.[50] Like in our case, developing dedicated techniques turns out beneficial.[51]

All in all, the tutorial fills a gap in the prior work on dependability of routing protocols. Compared to traditional formal approaches to proving properties of such protocols, it has a unique focus on dynamic behaviors. It also complements verification approaches based on automated tools, by offering techniques that enable generalizing their results. Finally, recognizing that there is no "silver bullet" in dependability, it aims to facilitate testing, implementation, and specification of a protocol by providing means for deriving precise formulations of properties that components of the protocol have to exhibit to guarantee particular behaviors in specific scenarios; these properties can be utilized to develop test

cases, implementations of the protocol, and, above all, its specification.

Overview of RPL

As a running example of a routing protocol for the tutorial, we adopt RPL. As mentioned previously, RPL is a recognized, mature, practical solution, which in addition exemplifies many typical consequences of standardization. Therefore, let us first give a brief overview of the protocol, notably the terminology it employs. The details can in turn be found in its main[3] and companion RFCs.

Scope of interest

Given RPL's complexity, in the tutorial we focus on the algorithms that constitute its foundation, enabling so-called upward routing. To explain, RPL supports any-to-any communication but emphasizes multipoint-to-point (convergecast) traffic, where many source nodes, normally low-power wireless devices, collaboratively forward their packets, using distance-vector routing, to a common destination node, typically a more powerful border router. This distance-vector routing in a so-called *upward* direction is utilized not only for convergecast but also for collecting topology information at the border router or the intermediate forwarding nodes themselves. The latter enables point-to-multipoint traffic in the opposite, so-called *downward*, direction: either by the border router initiating source routing (i.e. computing entire routes and embedding them into packets), or the intermediate nodes doing simple forwarding based on the data collected earlier, or a combination of the two approaches. Finally, point-to-point communication is obtained by first forwarding a packet upward to a border router (or a node with relevant topology information) and then redirecting it downward to the destination node.

Upward routing is thus fundamental to RPL: if it does not work correctly, downward routing also fails because of inconsistencies or a lack of topology information at the border router or the intermediate nodes. The algorithms enabling upward routing in RPL are also inherently decentralized and far more intricate than those enabling downward routing, which essentially boil down to periodically reporting topology information by all nodes and storing this information for use during forwarding packets downward. For these reasons, it is RPL's algorithms enabling upward routing that are our focus here. Since they follow the classic distance-vector approach, the techniques we present here can by and large be applied to other routing protocols following this approach.

Basic terminology

To support upward routing, each node keeps track of the wireless links to the other nodes in its radio range, so-called *neighbors*. RPL requires that the links be symmetric; asymmetric links are disregarded. Some of these links are selected to form *routing paths* to possible destination nodes. To minimize the memory and control traffic overheads due to maintaining these paths, RPL normally designates one or just a few nodes, typically only border routers, to act as upward destinations; other nodes can in turn be reached by downward routing from these destinations.

Routing paths are thus maintained per destination. To this end, each node is given a *rank* that describes a cost of reaching the destination from this node: the destination itself has the lowest rank, and the further away a node is from the destination, the higher its rank. Among the node's neighbors, those with their ranks lower than the node's own one are the node's so-called *parents*: forwarding a packet to a parent brings the packet closer to the destination. Normally, however, the node forwards all packets to a single parent, ideally the one with the lowest rank. Such a parent is called the node's *preferred parent*. From a global perspective, the nodes' links to parents should thus form a directed acyclic graph (DAG) with edges oriented toward the destination. Such a graph is referred to as a destination-oriented DAG, abbreviated as *DODAG*, whereas the destination node is called the DODAG *root*. The links to preferred parents, in turn, should form a directed tree that is a subgraph of the DODAG and has the DODAG root as its only sink. The paths in the tree are simply the upward routing paths that packets follow. Figure 1 illustrates these concepts.

Occasionally, a DODAG may need to be rebuilt. This entails all nodes forgetting their parents and ranks, and selecting new ones from scratch. Since such global rebuilding is not instantaneous, a node must be able to distinguish whether it has already performed the rebuilding locally or not. This is done with DODAG *versions*: each node remembers the DODAG's version and, when it observes a new version, it can transition to this version by reselecting its preferred parent and rank.

Finally, the costs of reaching the DODAG root, reflected in the nodes' ranks, can be measured by a range of *metrics*, such as the number of forwarding hops to the root, the estimated number of transmissions, or the average packet delivery latency.[52] Since different packets may need to be routed to minimize different costs, RPL defines so-called *instances*. An instance is an independent set of DODAGs, all optimizing the same cost. A node can belong to multiple instances. In each of them, it joins the defined DODAGs, ideally, their newest versions, for each choosing its rank and preferred parent independently.

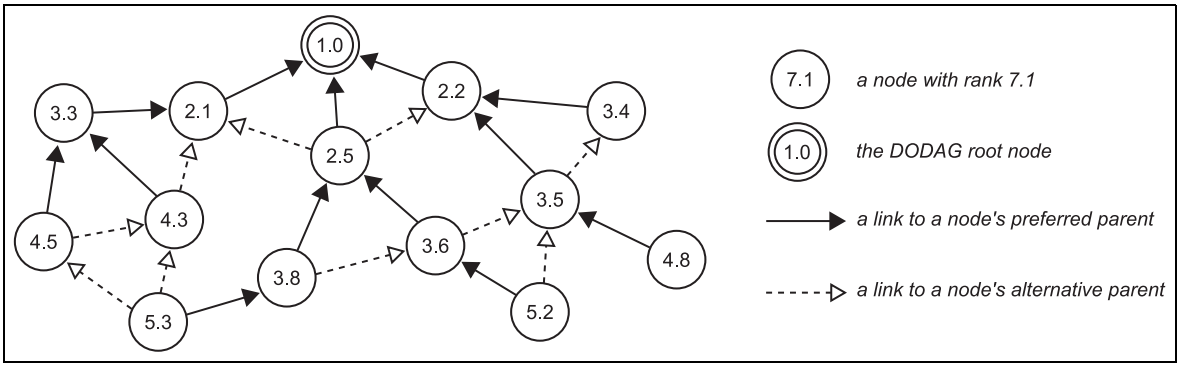

Figure 1. An example of a DODAG.

Each packet is in turn assigned by the source node to an appropriate instance,[53] depending on the cost metric that should be minimized when routing the packet.

Control traffic

A DODAG is built and maintained with two types of link-local ICMPv6 messages: DODAG Information Objects (*DIO*s) and DODAG Information Solicitations (*DIS*es). A DIO, transmitted by a node, advertises a path from this node to the root in a given DODAG version. Among others, it thus contains the DODAG's version and the node's rank. It can be sent either to a multicast IPv6 address denoting all neighbors of the sender or to a unicast address of a particular neighbor. In contrast, a DIS is used to solicit DODAG information by a node from its neighbors. It can optionally be restricted to a given RPL instance, a given DODAG, or even a particular version of the DODAG. Like a DIO, it can be multicast to all of the sender's neighbors or unicast to a particular one.

For a given node, the transmission of DIOs for a DODAG is by and large governed by a so-called Trickle timer.[54] In essence, it is an aperiodic timer whose goal is minimizing control traffic when the DODAG is stable, yet allowing for quickly reacting to changes and gradually stabilizing again. When the timer fires, the node multicasts a DIO to its neighbors. It can also unicast a DIO to a particular neighbor in response to a DIS from this neighbor. DISes, in turn, are typically multicast periodically, only when a node does not belong to any DODAG or even does not know its neighbors. Otherwise, unicast DISes to a specific neighbor may also be utilized by node to probe whether the neighbor is still alive and reachable. Such probing is not mandatory, though.

Apart from DIOs and DISes, RPL's core specification defines the following ICMPv6 messages: Destination Advertisement Objects (*DAO*s), Destination Advertisement Object Acknowledgements (*DAO-ACK*s), and Consistency Checks (*CC*s). DAOs and DAO-ACKs are utilized for reporting network topology upward to enable downward routing. CCs, in turn, are security-oriented messages, protecting against reply attacks and synchronizing cryptographic counters. This functionality assumes that upward routing works correctly, thereby being beyond our scope of interest.

Rank and parent selection

Based, in particular, on DIOs received for a DODAG, each node maintains a local *neighbor set*. An entry in the set corresponds to one of the node's neighbors and contains the neighbor's address and its last advertised DODAG version and rank. It may also contain the routing metric values for the neighbor and/or for the link with the neighbor, or other data that allows the node to compute these values. The neighbor set is thus the node's local view of its neighborhood.

From among the entries in its neighbor set, a node selects its preferred parent and computes its rank, depending on the parent's rank and metric values. This may happen either immediately whenever the neighbor set is updated or be deferred to allow for processing multiple updates in one batch. The details, however, are not part of RPL's main standard because optimizing different routing costs may require different metrics[52] and different ways of selecting parents and computing ranks. Instead, RPL delegates parent and rank selection to so-called *objective functions*, providing only constraints on how these functions must operate and guidelines on how they can be designed. Importantly, for every objective function, a node's rank must be greater than the node's preferred parent's at least by a constant, denoted as *MinHopRankIncrease*; the root's rank must in turn be exactly *MinHopRankIncrease*. Furthermore, in any DODAG version, a node's rank must not grow from its minimal value by more than another constant, *MaxVersionRankIncrease*; if it were to grow more, the node must adopt an infinite rank and a null parent, effectively losing its routing path to the root. Apart

from these constraints, however, there is a lot of flexibility in how an objective function can select parents and ranks, what costs it can optimize, and what metrics it can use.

The two commonly implemented objective functions are the Objective Function Zero (*OF0*)[55] and the Minimum Rank with Hysteresis Objective Function (*MRHOF*).[56] OF0 utilizes an adapted hop count as the underlying routing metric and selects as a node's preferred parent a neighbor that offers the node the lowest rank. MRHOF, in turn, is typically implemented for a range of routing metrics, notably the estimated transmission count, and, when selecting a node's preferred parent, avoids switching the current one if the gain in the node's rank were to be lower than a threshold, denoted *ParentSwitchThr*. This mechanism of not switching the parent unless sufficiently beneficial is called *hysteresis* and aims to make the DODAG more stable.

Open issues

As a final remark, despite specifying the core functionality associated with DODAG maintenance, the suite of documents constituting RPL's standards leaves a number of issues open to implementations. For instance, as we have already mentioned, when reselecting a node's preferred parent and rank, an implementation may choose an immediate or deferred approach. Similarly, it may incorporate virtually any policies on introducing new DODAG versions and transitioning from one version to another. The same is true for managing RPL's instances and many other protocol aspects.

Furthermore, some issues are delegated to external solutions, whose operation again need not be specified precisely. A prominent example is so-called *routing adjacency maintenance*, that is, the maintenance of the nodes' neighbor sets, notably routing metric values and reachability information. Although to some extent this is or can be done through DIOs and DISes, RPL's specification suggests other mechanisms, such as link-layer triggers[57] and IPv6 neighbor unreachability detection.[58] In either case, virtually no requirements are provided for such mechanisms.

In general, for a protocol like RPL, leaving issues underspecified is likely unavoidable. This, however, poses problems when implementing or modeling the protocol.

Linear temporal logic

Proving the behavior of a routing protocol such as RPL necessitates a formalism that, on one hand, guarantees that any derived properties indeed hold given the assumptions made in the proofs and, on the other hand, is powerful enough to enable deriving properties that are meaningful in practice. The formalism underlying this tutorial is LTL,[10] which we briefly introduce next, taking a perspective that in our view helps understanding the core parts of the tutorial and appreciating its soundness. Note that, by necessity, our discussion of LTL focuses only on the aspects relevant to the rest of the tutorial; fully mastering the formalism may in turn require some effort. To this end, we assume that the readers are familiar with propositional logic, on which LTL is based. Should the readers need more information on either of the formalisms, we recommend a classic book by Ben-Ari.[11]

Syntax and semantics

LTL extends propositional logic with the ability to express and prove temporal properties. To this end, it enriches the set of propositional operators that can be used in formulas (i.e. *true*, *false*, $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$) with new, temporal ones. The two fundamental temporal operators are $\bigcirc$ (named *next*) and $\mathscr{U}$ (referred to as *until*). However, others, which can be defined in terms of these two, are also commonly utilized for convenience, in particular $\Box$ (*always*/*globally*) and $\Diamond$ (*eventually*/*finally*).

The semantics of LTL is in turn defined for *computations*. A computation, σ, is an infinite sequence of states, numbered 0, 1, 2, ..., which models the flow of time. A formula, ϕ, may be satisfied in some state i of σ and not be satisfied in another state j, which is denoted $\sigma, i \models \phi$ and $\sigma, j \not\models \phi$, respectively. Each state of a computation is thus mapped to a set of formulas that are satisfied in this state. The temporal operators give the ability to express dependencies between states. By convention, it is assumed that a formula, ϕ, is satisfied for a given computation, σ, which is written $\sigma \models \phi$ (i.e. without the state number), if it is satisfied in the first state of σ, that is, $\sigma, 0 \models \phi$.

More formally, let AF be a countable, possibly infinite set of *atomic formulas*, that is, ones including neither the propositional nor temporal operators. In essence, it describes basic properties of a program at various moments, such as the memory contents, the instruction to be executed by a process, and the like. A computation is a function, $\sigma : \mathbb{N} \rightarrow 2^{AF}$, that maps each state $i \in \mathbb{N}$ to a set of (composite) formulas, $\sigma(i) \in 2^{AF}$ satisfied in this state, which are built out of the atomic formulas and the operators. More specifically, the satisfaction relation, $\models$, for a formula in any state i of any computation σ is defined by induction on the structure of the formula as follows:

- For an atomic formula $p \in AF$:

$\sigma, i \models p$ if and only if (abbreviated as *iff*) $p \in \sigma(i)$.

- For the *false* formula:

$\sigma, i \not\models false$.

- For a formula built with the propositional operator *implies*:

$\sigma, i \models \phi \rightarrow \psi$ iff $\sigma, i \not\models \phi$ or $\sigma, i \models \psi$.

- For a formula built with the temporal operator *next*:

$\sigma, i \models \bigcirc\phi$ iff $\sigma, (i + 1) \models \phi$.

- For a formula built with the temporal operator *until*:

$\sigma, i \models \phi \mathcal{U} \psi$ iff there exists some $j \geqslant i$, such that $\sigma, j \models \psi$ and for all $i \leqslant k < j$, $\sigma, k \models \phi$.

The semantics of the other propositional and temporal operators can be derived based on syntactic equivalences: $\neg\phi \equiv \phi \rightarrow false$, $true \equiv \neg false$, $\phi \vee \psi \equiv \neg\phi \rightarrow \psi$, $\phi \wedge \psi \equiv \neg(\neg\phi \vee \neg\psi)$, $\phi \leftrightarrow \psi \equiv (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$, $\Diamond\phi \equiv true \mathcal{U} \phi$, $\Box\phi \equiv \neg\Diamond(\neg\phi)$. In particular, for the aforementioned popular temporal operators, the semantics derived in such a way is intuitive:

- For a formula built with the temporal operator *eventually/finally*:

$\sigma, i \models \Diamond\phi$ iff there exists some $j \geqslant i$, such that $\sigma, j \models \phi$.

- For a formula built with the temporal operator *always/globally*:

$\sigma, i \models \Box\phi$ iff for all $j \geqslant i$, $\sigma, j \models \phi$.

Figure 2 gives a graphical illustration summarizing the semantics of the temporal operators.

To facilitate developing even more intuition, let us also consider two combinations of the operators that are commonly encountered in this tutorial. First, formula $\Diamond\Box\phi$ (*eventually always* ϕ) is satisfied in state i of a computation iff ϕ is satisfied in some state $j \geqslant i$ and all subsequent states of the computation, that is, iff starting from some state j, ϕ is continuously satisfied. Second, formula $\Box\Diamond\phi$ (*always eventually* ϕ) is satisfied in state i of a computation iff for every state $j \geqslant i$, there exists some state $k \geqslant j$, such that ϕ is satisfied in state k, that is, iff starting from state i, ϕ is satisfied repeatedly but not necessarily continuously. The two formulas are thus not equivalent. More specifically, the first implies the second but not the other way around.

Last but not least, let us exemplify formula patterns for two types of properties—*safety* and *liveness*—that are of particular importance in program verification. A safety property describes that some undesired effect (i.e. "something bad") never happens. A formula for such a property thus often has the following form: $\Box\neg\phi$, where ϕ describes the undesired condition. A liveness property, in turn, expresses that some desired effect (i.e. "something good") eventually occurs, typically under some condition. A formula corresponding to such a property is thus often constructed as follows: $\Box(\psi \rightarrow \Diamond\phi)$, where ψ and ϕ describe, respectively, the precondition and the desired effect. Put differently, safety properties correspond to invariants whereas

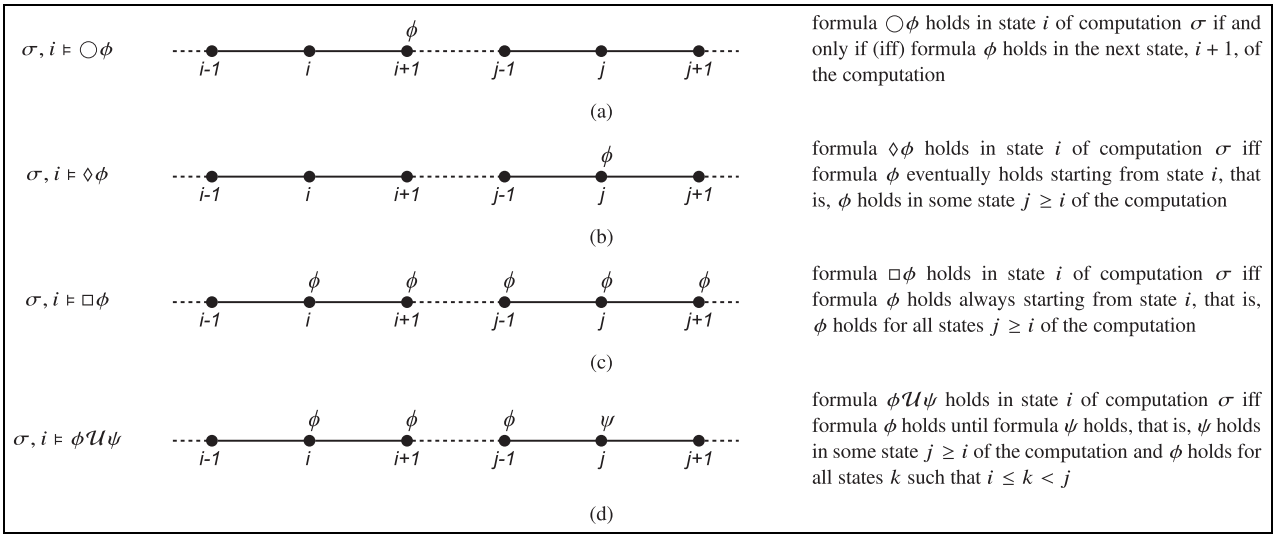

Figure 2. An illustration of the semantics of common temporal operators in LTL: (a) temporal operator *next*, (b) temporal operator *eventually*, (c) temporal operator *always*, and (d) temporal operator *until*.

```
   process P;                          process Q;
   var                                 var
     s : integer = 0;                    r : integer = 0;
     i : integer = 0;                    i : integer = 0;
     x : integer;                        y : integer;
   begin                               begin
1    while i < N do begin          1     while i < N do begin
2      x := random(M);             2       no_op();
3      send(Q, x);                 3       y := receive(P);
4      s := s + x;                 4       r := r + y;
5      i := i + 1;                 5       i := i + 1;
6    end;                          6     end;
7    while true do begin           7     while true do begin
8    end;                          8     end;
   end.                                end.
```

Listing 1. An example of a concurrent program.

liveness properties describe progress. To behave correctly, a program normally has to ensure both.

Modeling a program in LTL

To illustrate how LTL can be utilized for modeling and verification of concurrent programs, let us consider an example of such a program, presented in Listing 1. It consists of two processes: P and Q. Process P executes N iterations of a loop, with variable i being the iteration counter and N being a constant. In each iteration, it draws a random number, x, from the set $\{0, 1, \ldots, M-1\}$ (line 2), where M is a constant, sends the random value to process Q (line 3), and adds it to an accumulator variable, s (line 7–8). The infinite loop at the end (lines 2–2) is just to simplify verification in that we need not consider what happens when the process ends. Similarly, process Q executes N iterations of a loop, also with variable i being the iteration counter. In each iteration, it receives a number, y, from process P (line 4), and adds it to an accumulator variable, r (line 4). Again, the infinite loop at the end is just for simplicity. Likewise, the empty operation, no-op (line 2), is solely to make the numbers of similar lines in the two processes match to facilitate our presentation.

Assuming that communication provided by the *send* and *receive* functions is reliable, when both processes enter their infinite loops (lines 7–8), their accumulator variables should be equal. In other words, for any execution of the program, s is *eventually always* equal to r or, in the LTL notation, for every computation, σ, representing an execution of the program, we have: $\sigma \models \Diamond\Box(s = r)$. Our goal is to illustrate how this intuitive liveness property can be proved formally.

To this end, let us first explain how the program from Listing 1 can be modeled in LTL. It consists of the two processes that execute independently, that is, each process has its own address space and program counter. In the case of address spaces, what matters in the code displayed in the listing is the values of variables i, x, s for process P and i, y, r for process Q. To avoid ambiguities when referring to variable i, we will use a subscript denoting the process owning the variable: i_P for process P and i_Q for process Q. We will write $x = v$ and $i_Q = 2$ to express that P's variable x has value v and Q's variable i has value 2 respectively. To avoid constraining our reasoning to a particular hardware architecture, we assume that the *integer* type describes all integer values. From the perspective of program counters, in turn, we assume for simplicity that each numbered line of the listing is a single instruction. If this were not the case, we would consider lower-level machine code instead of the code from the listing, which would only add complexity without affecting our conclusions. We will write $@_P = 7$ to denote that P's program counter points at the instruction in line 7 of the code for P, that is, that this instruction is the next to be executed by P. Likewise, we will write $@_Q = 3$ to denote that process Q is about to execute its instruction in line 3.

For advancing the program counters, we assume a so-called *asynchronous process execution model* from distributed systems. This model is highly general because it does not restrict the relative running speeds of processes: in a period when one process executes one instruction, another process can execute an arbitrary, albeit finite number of instructions; what is more, these speeds can change arbitrarily in time. Such extremely weak assumptions thus allow for applying the model to virtually any distributed system. From our perspective, its main implication is that any instruction pointed by P's and Q's program counters *eventually* has a chance to be executed, which usually (but not always) advances the counter, as we formalize for the individual instructions shortly.

Likewise, we adopt an *asynchronous communication model*: when invoking the *send* function for a message (line 3), P never blocks but continues immediately and the message can take arbitrarily long to be delivered to Q. Moreover, since we assumed that the communication is reliable, Q is guaranteed to get the message, provided that it has a chance to do so, that is, it executes the *receive* operation (line 3) sufficiently many times (which it does). If no message has been sent by P, the *receive* function at Q waits. We can model such communication with a delivery multiset, $D_{P\to Q}$, that contains messages in transit from P to Q: they have been sent by P but not yet received by Q. $D_{P\to Q}$ is a multiset rather than a plain set because the *random* function (line 2) can return the same value more than once, and hence multiple instances of the value can be simultaneously in delivery.

This brings us to the random number generator. We simply assume that the invocation of the *random* function always eventually finishes and returns an arbitrary value from the subset $\{0, 1, 2, \ldots, M-1\}$. In other words, like previously, our assumptions on the random number generator are extremely weak and thus model virtually any solution employed in practice.

Given this information, we are ready to formalize a single LTL state of the considered system and possible transitions between such states that may happen during computations. More specifically, a state consists of: the values of P's and Q's program counters ($@_P$ and $@_Q$) and variables (i_P, i_Q, x, y, s, r), and the multiset representing messages in transit from P to Q ($D_{P\to Q}$). In the initial state, numbered 0, the program counters point to the first instructions of the processes (i.e. $@_P = 1$ and $@_Q = 1$), variables i_P, i_Q, s, r are equal to 0, variables x and y have undefined values, and the delivery multiset is empty (i.e. $D_{P\to Q} = \emptyset$). In the case of state transitions, in turn, we want to model as fine-grained changes to these elements as possible, given the selected granularity of instructions. Therefore, a state transition corresponds to executing exactly one instruction by one of the processes, which is a standard way of modeling a computation in LTL.[11] The possible state transitions due to executing instructions by process P are thus as follows (cf. also Figure 3 for a control flow diagram):

⟨P1⟩ The effect of executing the instruction in line 1 (i.e. when $@_P = 2$) depends on the value of i_P. If this value is equal to or greater than N, then $@_P$ becomes 2; otherwise, $@_P$ becomes 2. In both cases, the remaining state elements are unaffected.

⟨P2⟩ The instruction in line 2 assigns some value, $v \in \{0, 1, \ldots, M-1\}$, being the result of the *random* function, to variable x and advances $@_P$ to 2 without changing the other elements.

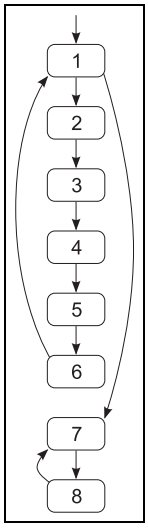

Figure 3. A control flow diagram for process P and Q from Listing 1 (the boxes correspond to the lines of code).

⟨P3⟩ Line 3 corresponds to sending a message, which results in adding the value of x (i.e. v) to the $D_{P\to Q}$ multiset, updating $@_P$ to 2, and leaving the other elements intact.

⟨P4⟩ Assuming that the value of s is u, executing the instruction in line 4 makes s equal to the sum of u and v (where v is the value of x), advances $@_P$ to 5, and changes no other elements.

⟨P5⟩ The instruction in line 5 increments the value of i_P by one, does the same for $@_P$, changing it to 6, and has no other effects.

⟨P6⟩ The only effect on the state of executing the instruction in line 6 is that $@_P$ becomes 1.

⟨P7⟩ Executing the instruction in line 7 just advances $@_P$ to 7 with no other effects.

⟨P8⟩ Finally, the instruction in line 8 sets $@_P$ back to 8, leaving the other elements intact.

For process Q, in turn, thanks to the appropriate line numbering in the listings, the transitions are mostly analogous. Let us thus give only the differing ones:

⟨Q1⟩ The only effect of executing the no-op instruction in line 2 is advancing $@_Q$ to 3.

⟨Q2⟩ The instruction in line 3 is receiving a message. Therefore, it can be executed only if there is some message in transit, that is, $D_{P\to Q} \neq \emptyset$. Let $v \in D_{P\to Q}$ be one of such messages (where $v \in \{0, 1, \ldots, M-1\}$). In such a case, $@_Q$ becomes 4, y becomes v, and v is also removed from $D_{P\to Q}$. The other state elements remain unmodified. If, in turn, there are no messages in delivery, that is, $D_{P\to Q} = \emptyset$, the instruction is not executed: no

state transition corresponding to process Q executing the instruction is allowed.

In summary, the model is rather intuitive. Nevertheless, it describes precisely what comprises our system from the verification perspective and how the system can evolve.

Verifying the program in LTL

The model is in principle what many automated model checkers (e.g. the aforementioned SPIN[39]) would use internally, likely in an optimized form, for the program from Listing 1. We could also feed such a tool directly with our target formula. In essence, the model checker would start from the initial state, and then by performing one of the allowed transitions, it would modify elements of this state, thereby obtaining a new state. By conducting such a state space exploration, it could check whether the formula holds, that is, it would produce and check all relevant states reachable from the initial one so as to ensure that the formula indeed holds in all possible executions of the program. All in all, using automated model checking for the program from Listing 1, we could attempt to automatically verify whether the property expressed as the aforementioned formula holds for the model describing the system running the program.

However, such a verification attempt would yield limited results because, as mentioned previously, to test if the formula is true, a model checker would enumerate all relevant states, which, depending on constants M and N, could lead to the aforementioned state space explosion. To give an example, in our model, the *random* function returns an arbitrary integer value between 0 and $M-1$. As the random number generator samples the M values with replacement (i.e. any value can be returned more than once), invoking the function N times during a computation already generates M^N possible computation branches that have to be verified. What is more, each of them branches further, depending on the interleaving of concurrent instructions of the two processes and the particular order of message receptions. Let us consider just those computations in which all messages are first sent by P, and only then is any message received by Q. For N messages in transit, assuming for simplicity that they are all unique, there are $N!$ different orders in which these messages can be received. In short, the number of possible computations just for this family of interleavings is another fast growing function of N. This implies that in practice our verification would have a chance to complete only if tiny values of M and N were selected.

What is more, its results would be valid only for the selected values of the constants. In contrast, formally proving that the results also hold for all $M>0$ and $N>0$ (i.e. generalizing the results) would anyway need to be done by other means. Although one may argue that, in the case of the considered program and formula, such generalization is intuitive, this is only because of their simplicity. For more complex systems, such as those describing routing protocols, proving properties, even seemingly intuitive ones, often turns out far from trivial and may lead to unexpected observations, as we will demonstrate in this tutorial. In other words, to overcome the limits of automated verification, one does need alternative approaches to formally proving LTL formulas.

One approach to deriving such proofs is to directly employ the semantics of LTL. In this approach, one analyzes a model of the considered system to show that the target formula is true or to find a counterexample. Depending on the formula, this may require demonstrating the existence of a particular computation in which the formula is true or proving that the formula holds in all computations satisfying some constraints. The process typically boils down to analyzing the initial state and possible state transitions. As such, it resembles automated model checking but does not require enumerating all states. Instead, it allows for using regular reasoning techniques, notably mathematical induction, to prove formulas generally, not just for specific configurations. For example, in the case of a routing protocol, one can prove in this way that some property holds in any network and not just the particular one that is fed to a model checker.

Another approach that in principle can give the same effects is to derive proofs in a deductive system for LTL. One such system is dubbed $\mathscr{L}$.[11] It comprises a set of axioms, that is, an initial set of formulas that are assumed to be true, and two inference rules, generalization and modus ponens, that transform formulas constituting premises into a formula being a conclusion. In this system, proving a formula is somewhat mechanical: a proof is a finite sequence of formulas, such that the last formula is the target formula and each formula is either an axiom or a result of applying generalization or modus ponens to respectively one or two earlier formulas. The axioms utilized in the proof can be either basic general formulas, defined as true by the system itself, or formulas specific to the program for which the proof is derived, essentially describing state transitions such as those we modeled previously. As to the inference rules, apart from generalization and modus ponens, derivative rules can also be used, notably induction. Importantly, $\mathscr{L}$ is sound and complete,[11] so it can be used instead of or in addition to the approach relying on the semantics.

In practice, to prove a given formula, usually the most fitting approach is selected. Consequently, to prove our formula, $\Diamond\Box(s = r)$, for the program from Listing 1, we will combine both approaches. To this

end, let $COUNT(D_{P\to Q})$ denote the number of messages in delivery, that is, the number of elements in multiset $D_{P\to Q}$. Similarly, let $SUM(D_{P\to Q})$ be the sum of the elements. Given these definitions, we could start by semantically proving the following safety formula, which may be treated as an additional lemma (i.e. a derivative formula in L). What the formula states is that if there are no messages in delivery, then the sum of the values in delivery is zero

$$\Box(\ COUNT(D_{P\to Q}) = 0 \rightarrow SUM(D_{P\to Q}) = 0\ ) \qquad (1)$$

However, since this formula is straightforward, we omit its proof, thereby treating it as an axiom. Instead, we focus on two more elaborate invariants. Their goal is to bind the messages in transit from P to Q with the values of either s and r or i_P and i_Q. More specifically—intuitively—r can lag behind s in that the value of r should always be less than s by the sum of all messages that have been sent by P but not yet received by Q, $SUM(D_{P\to Q})$; in particular, if there are no such messages, r should be equal to s. We could express this with the following formula: $\Box(s = r + SUM(D_{P\to Q}))$. Likewise, i_Q lags behind i_P by the number of messages in delivery, which we could formalize: $\Box(i_P = i_Q + COUNT(D_{P\to Q}))$. However, this intuition is not completely valid, as there may be inconsistencies between the moment when a message is sent or received and the corresponding moment when s/i_P or r/i_Q is incremented with $x/1$ or $y/1$, respectively. The correct versions of the formulas thus have to take this into account, thereby depending on the current instruction to be executed by each of the two processes. The revised formula for s and r is thus as follows

$$\begin{array}{lllll}\Box(& (@_P \neq 4 \wedge @_Q \neq 4 & \rightarrow & s & = r + SUM(D_{P\to Q})) \wedge \\ & (@_P = 4 \wedge @_Q \neq 4 & \rightarrow & s + x & = r + SUM(D_{P\to Q})) \wedge \\ & (@_P \neq 4 \wedge @_Q = 4 & \rightarrow & s & = r + SUM(D_{P\to Q}) + y) \wedge \\ & (@_P = 4 \wedge @_Q = 4 & \rightarrow & s + x & = r + SUM(D_{P\to Q}) + y)\ )\end{array} \qquad (2)$$

The formula for i_P and i_Q is similar

$$\begin{array}{ll}\Box(& ((@_P \neq 4 \wedge @_P \neq 5) \wedge (@_Q \neq 4 \wedge @_Q \neq 5) \rightarrow \\ & \quad i_P = i_Q + COUNT(D_{P\to Q})) \wedge \\ & ((@_P = 4 \vee @_P = 5) \wedge (@_Q \neq 4 \wedge @_Q \neq 5) \rightarrow \\ & \quad i_P + 1 = i_Q + COUNT(D_{P\to Q})) \wedge \\ & ((@_P \neq 4 \wedge @_P \neq 5) \wedge (@_Q = 4 \vee @_Q = 5) \rightarrow \\ & \quad i_P = i_Q + COUNT(D_{P\to Q}) + 1) \wedge \\ & ((@_P = 4 \vee @_P = 5) \wedge (@_Q = 4 \vee @_Q = 5) \rightarrow \\ & \quad i_P + 1 = i_Q + COUNT(D_{P\to Q}) + 1)\ )\end{array} \qquad (3)$$

Proof of Formula (2). We utilize standard induction on state number.

As the inductive base, we consider the initial state 0, in which $@_P = 1$, $@_Q = 1$, $s = 0$, $r = 0$, $i_P = 0$, $i_Q = 0$, x and y are undefined, and $D_{P\to Q} = \emptyset$. Let us observe that the formula is constructed in such a way that the four implications constituting it refer to disjoint configurations of $@_P$ and $@_Q$: exactly one implication has its premises true. More specifically, since $@_P = 1$ and $@_Q = 1$ in state 0, only the premises of the first implication in the formula are true. However, also the conclusion of this implication, $s = r + SUM(D_{P\to Q})$, is true, because we have $s = 0$, $r = 0$, and $SUM(D_{P\to Q} = \emptyset) = 0$. All in all, since the other three implications are also true because of their premises being false, the formula as a whole is indeed true in state 0.

For the inductive step, we take an arbitrary state $t \in \{0, 1, 2, \ldots\}$ and—assuming that the formula is true in all states up to and including t—we prove that it is also true in state $t + 1$. We achieve this by analyzing all state transitions to show that none of them makes the formula false.

Let us start with transition $\langle$P1$\rangle$. Before the transition, we have the formula true and $@_P = 1$. (and hence $@_P \neq 4$). Consequently, we must have either $s = r + SUM(D_{P\to Q})$, if $@_Q \neq 4$, or $s = r + SUM(D_{P\to Q}) + y$, if $@_Q = 4$. After the transition, in turn, $@_P$ becomes either 2 or 7 (and hence still $@_P \neq 2$), while the other elements of the state, notably s, r, y, $D_{P\to Q}$, and $@_Q$, do not change. Therefore, since either $s = r + SUM(D_{P\to Q})$ or $s = r + SUM(D_{P\to Q}) + y$, depending on the value of $@_Q$, is true before the transition and none of the values changes, the same equality holds after the transition: the transition does not invalidate the formula.

Precisely the same reasoning can be applied to transitions $\langle$P2$\rangle$ and $\langle$P5$\rangle$ ... $\langle$P8$\rangle$.

Let us thus consider transition $\langle$P3$\rangle$. Before the transition, we have $@_P = 1$ (and hence $@_P \neq 4$). Let us also denote the values of the other variables as follows: $s = v_s$, $r = v_r$, $SUM(D_{P\to Q}) = v_{SD}$, and $x = v_x$, for some integers v_s, v_r, v_{SD}, v_x, and, if $@_Q = 4$, let $y = v_y$ for some integer v_y. Moreover, from the inductive assumption we know that either $s = r + SUM(D_{P\to Q})$, if $@_Q \neq 4$, or $s = r + SUM(D_{P\to Q}) + y$, if $@_Q = 4$. In other words, if $@_Q \neq 4$, then $v_s = v_r + v_{SD}$ ($\bigstar$); if in contrast $@_Q = 4$, then $v_s = v_r + v_{SD} + v_y$ ($\bigstar\bigstar$). After the transition, in turn, $@_P$ becomes 4, v_x is added to the $D_{P\to Q}$ multiset, and hence $SUM(D_{P\to Q})$ becomes $v_{SD} + v_x$, and the values of the other variables are unchanged, in particular $s = v_s$, $r = v_r$, $x = v_x$, and, if $@_Q = 4$, $y = v_y$. From the fact that now $@_P = 2$, to prove that the formula is true, we must show that either $s + x = r + SUM(D_{P\to Q})$, if $@_Q \neq 4$, or $s + x = r + SUM(D_{P\to Q}) + y$, if $@_Q = 4$. *Case 1*: Let $@_Q \neq 4$. We have $s + x = v_s + v_x$ as neither s nor x changes its values during the transition. We also have $r + SUM(D_{P\to Q}) = v_r + (v_{SD} + v_x)$ because r does not change its value and $SUM(D_{P\to Q})$ becomes $v_{SD} + v_x$. From ($\bigstar$), in turn, we know that equation $v_s = v_r + v_{SD}$ holds. Adding v_x to both sides of the equation and making use of the associativity of addition, we get

$v_s + v_x = v_r + (v_{SD} + v_x)$. In other words, indeed $s + x = v_s + v_x = v_r + (v_{SD} + v_x) = r + SUM(D_{P\to Q})$.
Case 2: Let $@_Q = 4$. We again have $s + x = v_s + v_x$ as neither s nor x changes its values during the transition. We also have $r + SUM(D_{P\to Q}) + y = v_r + (v_{SD} + v_x) + v_y$ because r and y do not change their values and $SUM(D_{P\to Q})$ becomes $v_{SD} + v_x$. From (★★), in turn, we know that equation $v_s = v_r + v_{SD} + v_y$ holds. Again, adding v_x to both sides of the equation and making use of the associativity and commutativity of addition, we get $v_s + v_x = v_r + (v_{SD} + v_x) + v_y$. Therefore, indeed $s + x = v_s + v_x = v_r + (v_{SD} + v_x) + v_y = r + SUM(D_{P\to Q}) + y$. The combined conclusions for Cases 1 and 2 thus indeed prove that also transition ⟨P3⟩ does not invalidate the formula.

The reasoning for transition ⟨P4⟩ is symmetric, so we leave it to the interested readers.

Furthermore, transitions ⟨P1⟩ and ⟨P4⟩ ... ⟨P8⟩ are the same as the corresponding transitions for process Q. Moreover, a similar reasoning as, for instance, for transition ⟨Q2⟩ can be applied to transition ⟨P5⟩, which represents a no-op. Therefore, what is left to analyze for process Q is the unique message reception transition, that is, transition ⟨Q3⟩.

Before transition ⟨Q3⟩, we have $@_Q = 3$ (and hence $@_Q \neq 4$), $s = v_s$, $r = v_r$, and $SUM(D_{P\to Q}) = v_{SD}$, for some integers v_s, v_r, v_{SD}, and, if $@_P = 4$, also $x = v_x$ for some integer v_x. In addition, from the inductive assumption, we have either $s = r + SUM(D_{P\to Q})$, if $@_P \neq 2$, or $s + x = r + SUM(D_{P\to Q})$, if $@_P = 4$. In other words, if $@_P \neq 4$, then $v_s = v_r + v_{SD}$ (‡); otherwise, if $@_P = 4$, then $v_s + v_x = v_r + v_{SD}$ (‡‡). After the transition, in turn, $@_Q$ becomes 4, y becomes v_y for some integer $v_y \leqslant v_{SD}$, $SUM(D_{P\to Q})$ becomes $v_{SD} - v_y$, and the other variables preserve their values. Since now $@_Q = 4$, to prove that the formula is true, we must show that either $s = r + SUM(D_{P\to Q}) + y$, if $@_P \neq 4$, or $s + x = r + SUM(D_{P\to Q}) + y$, if $@_P = 4$. *Case 1*: Let $@_P \neq 4$. We have $s = v_s$ as the value of s does not change during the transition. We also have $r + SUM(D_{P\to Q}) + y = v_r + (v_{SD} - v_y) + v_y$ as r does not change during the transition, $SUM(D_{P\to Q})$ becomes $v_{SD} - v_y$, and y becomes v_y. From (‡), in turn, we know that equation $v_s = v_r + v_{SD}$ holds. Subtracting and adding v_y to the right-hand side of the equation and making use of the associativity of integer addition, we get $v_s = v_r + (v_{SD} - v_y) + v_y$. Rewriting the three equations, we thus get the desired conclusion $s = v_s = v_r + (v_{SD} - v_y) + v_y = r + SUM(D_{P\to Q}) + y$.
Case 2: Let $@_P = 2$. We have $s + x = v_s + v_x$ as neither s nor x changes its value during the transition. We also have $r + SUM(D_{P\to Q}) + y = v_r + (v_{SD} - v_y) + v_y$ as r does not change during the transition, $SUM(D_{P\to Q})$ becomes $v_{SD} - v_y$, and y becomes v_y. From (‡‡), in turn, we know that equation $v_s + v_x = v_r + v_{SD}$ holds. Again, we can enhance the equation with v_y, thereby obtaining $v_s + v_x = v_r + (v_{SD} - v_y) + v_y$. Rewriting the three equations, we again get the desired conclusion $s + x = v_s + v_x = v_r + (v_{SD} - v_y) + v_y = r + SUM(D_{P\to Q}) + y$. The combined conclusions for Cases 1 and 2 thus indeed prove that also transition 2 does not invalidate the formula.

Having shown that no transition makes the formula false, we proved the inductive step, that is, that for any state t, if the formula is true in this state, then it is also true in the next state, $t + 1$. By applying mathematical induction to the base and the inductive step, we can thus conclude that Formula (2) is indeed an invariant of the program from Listing 1, which ends the proof.□

An analogous semantic proof can be conducted for Formula (3), so we leave it as an exercise for the interested readers. Instead, we exemplify how the deductive system, $\mathscr{L}$, can be used to complement semantic proofs. To this end, let us first introduce a few formulas that can be derived directly from the transitions for the considered program and will serve as axioms describing the control flow in the program. In essence, they express the global effects of particular instructions on the program counters and variables i_P and i_Q. More specifically, for any integer value v_i, the first formula states that whenever process P enters the infinite loop in lines 7–8 with i_P having value v_i, then the process remains in the loop forever without any further changes to i_P

$$\Box(((@_P = 7 \vee @_P = 8) \wedge i_P = v_i) \rightarrow \Box((@_P = 7 \vee @_P = 8) \wedge i_P = v_i)) \tag{4}$$

The same formula for process Q is as follows

$$\Box(((@_Q = 7 \vee @_Q = 8) \wedge i_Q = v_i) \rightarrow \Box((@_Q = 7 \vee @_Q = 8) \wedge i_Q = v_i)) \tag{5}$$

The next formula, in turn, describes the termination of the main loop of process P, that is, that starting from i_P equal to 0, the loop will eventually finish with i_P equal to N. Like the previous ones, the formula can be proved by mathematical induction, semantically or by using $\mathscr{L}$. Since it is straightforward, we omit the proofs for brevity and treat the formula as an axiom

$$\Box((@_P = 1 \wedge i_P = 0) \rightarrow \Diamond(@_P = 7 \wedge i_P = N)) \tag{6}$$

An analogous formula is true for process Q. Its proof is only slightly more involved in that it necessitates employing Formulas (3), (4), and (6) to show that in each iteration, the *receive* instruction from line 3 of Listing 1 is eventually executed as there is ultimately a message to be received. Therefore, again, we omit the proof of the formula for brevity, treating the formula as an axiom

$$\Box((@_Q = 1 \wedge i_Q = 0) \rightarrow \Diamond(@_Q = 7 \wedge i_Q = N)) \tag{7}$$

We are now ready to prove our main hypothesis: for any computation, σ, of the program from Listing 1, we have $\sigma \models \Diamond\Box(s = r)$. Because of space constraints, we do not quote here all basic general axioms of propositional logic and the deductive system, $\mathscr{L}$, that are utilized in the proof. However, since they are intuitive, we are still able to precisely present the major steps of the proof.

Proof. Combining Formulas (4)–(7), we get

$$\Box((@_P = 1 \wedge @_Q = 1 \wedge i_P = 0 \wedge i_Q = 0) \rightarrow \Diamond\Box((@_P = 7 \vee @_P = 8) \wedge (@_Q = 7 \vee @_Q = 8) \wedge i_P = N \wedge i_Q = N))$$

In other words, if the two processes start from their first instructions ($@_P = 1 \wedge @_Q = 1$) with their i variables equal to the initial values ($i_P = 0 \wedge i_Q = 0$), then eventually they will reach and forever remain in their infinite loops (($@_P = 7 \vee @_P = 8) \wedge (@_Q = 7 \vee @_Q = 8$)) with their i variables being constantly equal to N ($i_P = N \wedge i_Q = N$). Combining the formula with Formula (3), which expresses the invariant dependency between i_P, i_Q, and $COUNT(D_{P\rightarrow Q})$, we can infer that

$$\Box((@_P = 1 \wedge @_Q = 1 \wedge i_P = 0 \wedge i_Q = 0) \rightarrow \Diamond\Box((@_P = 7 \vee @_P = 8) \wedge (@_Q = 7 \vee @_Q = 8) \wedge COUNT(D_{P\rightarrow Q}) = 0))$$

Likewise, applying also the invariant for $COUNT(D_{P\rightarrow Q})$ and $SUM(D_{P\rightarrow Q})$, that is, Formula (1), we prove that in addition $SUM(D_{P\rightarrow Q})$ is eventually always zero

$$\Box((@_P = 1 \wedge @_Q = 1 \wedge i_P = 0 \wedge i_Q = 0) \rightarrow \Diamond\Box((@_P = 7 \vee @_P = 8) \wedge (@_Q = 7 \vee @_Q = 8) \wedge SUM(D_{P\rightarrow Q}) = 0))$$

Therefore, combining this with the invariant describing the dependency between $SUM(D_{P\rightarrow Q})$, s, and r, that is, Formula (2), we can conclude that the following formula is true

$$\Box((@_P = 1 \wedge @_Q = 1 \wedge i_P = 0 \wedge i_Q = 0) \rightarrow \Diamond\Box(s = r))$$

Finally, since any computation, σ, of the program starts with a state in which the program counters of the two processes point to the first instructions and the variables are equal to their initial values (i.e. we have $\sigma \models @_P = 1 \wedge @_Q = 1 \wedge i_P = 0 \wedge i_Q = 0$), we can conclude from the formula that in the computation s is eventually always equal to r (i.e. $\sigma \models \Diamond\Box(s = r)$), which ends the proof. □

All in all, we hope to have illustrated that LTL is a sound and powerful formalism. It allows for capturing many intricacies of the dynamic behavior of concurrent programs. We will demonstrate its potential for deriving practically relevant formal guarantees for a routing protocol, like RPL.

Modeling RPL's dynamic behavior

As we showed in the previous section, proving dynamic properties formulated in LTL for a concurrent program requires a model of a system running the program. In the case of a routing protocol, in particular RPL, the model should cover not only the algorithms constituting the protocol but also the impact of the environment in which they operate, notably the possible dynamics of nodes and links. The parts of the model describing the algorithms can be constructed based directly on algorithm descriptions, for example, the relevant RFCs in the case of RPL. They can also be built from existing protocol implementations, such as the aforementioned TinyRPL and ContikiRPL. The behavior of the environment, in turn, is typically modeled based on community knowledge.

In any case, the model has to satisfy two seemingly conflicting requirements. On one hand, its assumptions should not be oversimplified to a point that meeting them would be infeasible in the real world. Otherwise, the properties derived for the model would have little, if any practical relevance. On the other hand, its level of detail (i.e. complexity) should be under control. Otherwise, verifying even a simple property would be tedious, an example of which is arguably the proofs from the previous section. One of the main challenges in modeling is thus discovering which aspects can be simplified, how to perform the simplification, and what its practical consequences are.

A common first step to addressing this challenge is building a targeted model, including only those aspects of the protocol that are relevant to the behaviors of interest. If properly constructed, such a dedicated model does not preclude verifying other behaviors. On the contrary, this can be done by extending the model with new components or removing or replacing some existing ones.

Consequently, we will develop such a dedicated model here. More specifically, our model will target RPL's DODAG construction and maintenance, which is the enabler of upward routing. To further limit its complexity, we will focus on a single instance and DODAG, because considering more would bring little new insight from the perspective of the tutorial. In addition, having completed the tutorial, the interested readers will be in position to develop the model appropriately. As a side note, the model will be a union of the models from our previous papers,[6,7] further extended to cover extra aspects. This in particular

implies that its key parts have been double-checked against ContikiRPL and TinyRPL to ensure that they are consistent and implementable.

We start by defining a state of the considered system, as required by the notion of computation in LTL. We then proceed to identifying possible state transitions. Finally, we formalize axioms describing the interplay between the two, which determine the dynamic behavior of the system.

System state

In line with the common terminology and previous sections, a system running a routing protocol such as RPL is defined in terms of *nodes* and *links*. Nodes host processes that execute the program implementing the algorithms constituting the protocol. Links are directed logical connections between pairs of nodes, thereby being a medium through which the processes can exchange packets. Together, they form a fixed directed *communication graph*, $G_{COM} = (V_{COM}, E_{COM})$, in which the vertices correspond to the nodes (i.e. $V_{COM} = \mathbb{N}\text{odes}$), whereas the edges correspond to the links (i.e. $E_{COM} = \mathbb{L}\text{inks}$). We assume that the number of nodes (and hence links) is finite and the graph is connected. These two assumptions are necessary for liveness properties.

Both nodes and links can be subject to failures, which disrupt their regular operation. A basic failure class in distributed systems is so-called *crash-stop failures*.[12] When a node crashes, it forever stops executing its program. Likewise, when a link crashes, it forever stops delivering packets. In contrast, the broadest class represents so-called *Byzantine failures*, in which failing nodes and links may behave arbitrarily and may even collude. Yet, hardly any practical routing protocol—RPL not being an exception—can tolerate nodes failing that way: routing protocols normally assume collaborative rather than malicious nodes. Similarly, nonmalicious links are normally considered.

Consequently, although one can adopt any failure class in LTL-oriented models, here we settle on a class that arguably covers a sufficient range of failures encountered in practice: so-called *crash-recovery failures*. In this class, a crashed node or link can recover: after recovery, a node starts its program from the beginning and a link resumes delivering packets. A node or link may thus alternate between being live and dead, which more faithfully models a real network. In this context, since the delivery of a packet between two neighbors depends on the liveness of these nodes and the links between them in some time span, to simplify formulations of our properties, we introduce a concept of "adjacency": we say that two nodes are *adjacent* in a given LTL state iff in this state they are both live and the links between them in both directions are live as well.

Finally, it is important to note that having the communication graph, G_{COM}, fixed does not preclude modeling dynamic networks. This is because the liveness of nodes and links may change dynamically, and so may other parameters affecting a DODAG, such as routing metrics of nodes and links and packet loss rates over individual links. In effect, even a highly mobile network can be modeled by having the graph contain all links and varying the liveness of individual links according to a particular mobility pattern. Given this introduction, we are ready to discuss what information an individual node and link contributes to a global system state.

Node state. Any global state has to contain for each node the data that the algorithms considered in the model require to run on this node. To this end, like in the example from the previous section, rather than modeling the node's memory bytes, processor registers, and the like, we will model only values of relevant program variables. This approach reduces complexity and ensures the correct types of these variables. At the same time, it does not preclude modeling practical problems, like those related to storage limitations, if we chose to do so. For instance, if we wanted to verify the behavior of a protocol under a lack of packet buffers, we could introduce a variable representing a limited-size buffer pool. In our case, however, since the model targets RPL's DODAG construction and maintenance, we limit our interest to the variables listed in Table 1 and corresponding to the concepts mentioned in the overview of the protocol.

Table 1. Variables comprising the state of a node.

Name	Description
Version	contains a nonnegative integer value that equals the current DODAG version the node hosting the variable belongs to
Prefpar	is the identifier of the neighbor that the hosting node has selected as its preferred parent in the current DODAG version or a special value, *null*, if the node has no parent
Rank	contains a nonnegative integer value that represents the hosting node's rank in the current DODAG version or a special value, infinity (∞)
Minrank	equals to the minimal rank that the hosting node has ever had in the current DODAG version, which in particular may be infinity
neighborset	is a set of the hosting node's neighbors of which the node is aware, with each entry, *n*, corresponding to one such neighbor and containing the fields listed in Table 2

DODAG: destination-oriented DAG.

Table 2. Fields of each entry, n, in a node's `neighborset`.

Name	Description
n. id	equal to the neighbor's identifier and used as the discriminator for the set
n. version	containing the DODAG version that the hosting node believes the neighbor currently belongs to
n. rank	equal to the rank in this DODAG version that the hosting node believes the neighbor currently has
n. reachable	being *true* if the hosting node believes that the neighbor is reachable (e.g. can be communicated with because the two nodes are adjacent) or *false* otherwise
...	possibly some other fields (e.g. with routing metric values or information relevant to unreachability detection) that are treated as "black boxes"

DODAG: destination-oriented DAG.

Table 3. RPL's constants a node utilizes.

Name	Description
InitialDODAGVersion	a value of the DODAG version a node initially has
MinHopRankIncrease	a minimal value by which a node's rank should differ compared to the parent's rank
MaxVersionRankIncrease	a maximal value by which a node's rank is allowed to grow in a given DODAG version
ParentSwitchThr	a minimal value by which a node's rank has to change in order to warrant preferred parent reselection (used for some objective functions)

DODAG: destination-oriented DAG.

Variables `version`, `prefpar`, and `rank` describe the node's place in a DODAG (version). Variable `minrank` is utilized to enforce that the node's `rank` never grows unbounded, that is, above `MaxVersionRankIncrease`. Finally, variable `neighborset` is the node's knowledge of its neighborhood, which is necessary, among others, when selecting *prefpar* and `rank`. Each entry of a node's `neighborset` contains the fields enumerated in Table 2.

For any node $X \in \mathbb{N}\text{odes}$ and any global state $t \in \{0, 1, 2, \ldots\}$ of the system in a specific LTL computation involving the model, we will use notation $var_X(t)$ to refer to the value of X's variable `var` in state t, for example, $minrank_X(t)$. Although in states in which the node is dead, we could adopt some placeholder values for its variables, to reflect the actual situation, we consider these values as undefined instead.

This brings us to another observation: any global state has to incorporate for each node information indicating whether the node is live or dead in this state. Accordingly, we will denote as $\mathbb{L}\text{ive}\mathbb{N}\text{odes}(t) \subseteq \mathbb{N}\text{odes}$ the set of nodes that are live in state t of a specific LTL computation. In particular, we assume that in the initial state, $t = 0$, all nodes are dead, that is, $\mathbb{L}\text{ive}\mathbb{N}\text{odes}(0) = \emptyset$.

Finally, we assume for simplicity that the identifier of the DODAG root node is predefined and does not change in any computation. If necessary, this assumption can easily be dropped, though. For completeness, in Table 3, we also provide a summary of constants that a node in RPL utilizes.

Link state. For each link, in turn, any global state has to contain packets that are in transit over the link in this state. Since link-layer communication between nodes is normally implemented by a stack of various low-lever protocols and operating system modules, being in transit may in practice describe several conditions. To give some examples, a packet in transit may be: in some buffer at some layer of the operating system of the transmitting node, in a buffer of the radio of this node, on air, in a buffer of the radio of the receiving node(s), or in one of the operating system buffers of these nodes. Consequently, to reduce the complexity of our model and at the same time cover all such situations, we represent all packets in transit over a given link like in the example from Listing 1: as a delivery multiset for this link, denoted `linkdset`.

Furthermore, as highlighted previously, the core of DODAG maintenance is done mostly through ICMPv6 DIO messages, which carry the necessary information; DIS messages, in turn, are used mostly to solicit transmissions of DIO messages, whereas the purpose of RPL's other messages is delivering functionality that already relies on upward routing, thereby being beyond the scope of our interest. Consequently, by properly abstracting the rules of DIO transmissions, neighbor reachability detection, and routing metric value maintenance—which we address shortly—we can limit our reasoning on DODAG construction to DIO messages. Accordingly, we assume that the delivery multiset, `linkdset`, for a link contains only packets carrying such messages, thereby ignoring other packets.

What is more, we will be interested in DIO messages that can be received by a specific node. Therefore, for convenience, for each node $X \in \mathbb{N}\text{odes}$, we define a per-node delivery multiset, $nodedioset_X$, as the union of the multisets for the incoming links to this node, that is, in any state $t \in \{0, 1, 2, \ldots\}$, we have $nodedioset_X(t) = \bigcup_{\langle Y,X\rangle \in \text{Links}} linkdset_{\langle Y,X\rangle}(t)$. All link-related variables contributing to the system state are summarized in Table 4.

Each DIO message, d, in our model includes the fields listed in Table 5. The fields simply allow the

Table 4. Variables comprising the state of a link.

Name	Description
Linkdset	is a multiset comprising all DIO messages in transit over the link, with each message containing the fields listed in Table 5
Nodedioset	defined for a convenience for each node X as a multiset containing all DIO messages in transit to the node, that is, *nodedioset* $_X = \bigcup_{\langle Y,X\rangle \in \mathbb{L}inks}$ *linkdset* $_{\langle Y,X\rangle}$

DIO: DODAG information objects.

Table 5. Fields of a DIO message, `d`, in a links's `linkdset`.

Name	Description
d. id	contains the transmitting node's identifier
d. version	equals to the node's DODAG version (at the time of the transmission)
d. rank	equals to the node's rank within this version (at the time of the transmission)
...	possibly some other fields (e.g. for neighbor reachability detection and routing metric value maintenance) that are again treated as "black boxes"

DODAG: destination-oriented DAG.

transmitting node to advertise the current values of its corresponding variables, which is in general the goal of a DIO message.

Finally, like a node, a link may be live or dead in any state. Therefore, we denote as $\mathbb{L}ive\mathbb{L}inks(t) \subseteq \mathbb{L}inks$ the set of links that are live in state t of a specific LTL computation. Again, we assume that in the initial state, $t = 0$, all links are dead, that is, $\mathbb{L}ive\mathbb{L}inks(0) = \emptyset$. In this context, unlike the previous variables, the values of *linkdset* or *nodedioset* are defined even if the corresponding link or node is dead. This is because these variables represent multiple situations a DIO message can be in and hence a failure of a single component need not affect the message. For example, the presence of a message in a delivery multiset can represent a situation where the message is still in some buffer of the transmitting node and hence need not be affected by the crash of a link or a receiving node. We will formalize precise message delivery guarantees shortly.

State transitions

To recap, in our model, a global state of the system incorporates information on which nodes and links are live, what packets are in transit over which links, and what values the local variables of the live nodes have. To be able to model an LTL computation, we need to define the possible transitions of the system between such states.

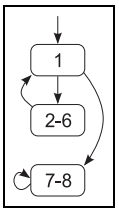

Figure 4. A compacted control flow diagram for process *P* and *Q* from Listing 1 (cf. Figure 3).

As a starting point, consider again the example from Listing 1 (with the control flow also visualized in Figure 3), in which a state transition was always due to one of the processes executing the instruction pointed by its program counter. Adopting that approach here would be problematic, though. Even if we provided code of a program implementing RPL, it would be rather voluminous, considering RPL's specification and its implementations. This would remain true even if we limited the program to the aspects relevant only to the modeled variables, which we have confirmed empirically. In effect, any computation for the model would involve an excessive number of states, a large fraction of which would be completely immaterial to our reasoning and, yet, would have to be taken into account in our analyses. As an illustration, consider for instance Formula (2) from the aforementioned example. Even though it describes a simple invariant between two variables, it is composed out of four implications, whose sole purpose is addressing states with arguably irrelevant, temporal inconsistencies between the variables.

This observation hints at a potential solution to modeling state transitions: rather than to an execution of an individual instruction, a state transition can correspond to an execution of a logically related group of program instructions. For instance, in the example from Listing 1, instead of modeling the control flow of processes P and Q as in Figure 3, we can model it by grouping logically related instructions as in Figure 4. In particular, having such a single composite instruction describes the execution of the entire body of the program loop (lines 2–6 of Listing 1) would reduce the elaborate invariant expressed in Formula (2) to the elegant and intuitive $\Box(s = r + SUM(D_{P\to Q}))$, which models the core dependency between variables s and r.

This approach has to be further adapted to capture not only the studied algorithms but also the impact the environment has on them. In particular, in the example from Listing 1, we disregarded process and communication channel failures. To address this issue, we can model a state transition as a more general *event*. An event can correspond to an execution of a logically related group of program instructions, at a granularity even coarser than in the previous example, or some

external phenomenon of the environment, including a crash or recovery of a processes or communication channel.

In general, apart from reducing model complexity, the event-based approach to describing state transitions has several advantages. It reflects the way protocol specifications are written, as they typically describe how protocols react to various events. It also matches many software architectures for low-power wireless systems, which are often event-driven. At the same time, without any special provisions, it can model asynchronous process execution and communication, which, as mentioned previously, are usually assumed for distributed algorithms. Likewise, with well-defined effects on the global state, it is by no means "less formal" than the program-counter-based approach from that section: events are simply higher level instructions that operate on the global state defined in the model.

Consequently, we employ the event-based approach here: a transition between two global system states is due to a particular event occurring. In the rest of this section, we thus list events triggered by the program running on the nodes and events caused by the environment. We give some intuition behind each event and discuss what components of the global system state it affects. The precise temporal properties for the events are in turn formalized in the next section.

Program-driven events. The events triggered by the program can occur only if the node they concern (i.e. the node executing RPL's software) is live. They are simply responsible for modifying the node's local variables and the delivery multisets of the links to and from the node, and are as follows:

(a) *DODAG version generation*—occurs when RPL decides to build a new DODAG version; causes the executing node to set its variable `version` to a new version number and to reset its variables `prefpar`, `rank`, and `minrank` to their initial values (as formalized in the next section);
(b) *DODAG version adoption*—takes place when RPL on the executing node decides to join a given DODAG version; again, causes the node to change its *version* to the given version number and to reset its variables `prefpar`, `rank`, and `minrank`;
(c) *Parent and rank reselection*—happens whenever RPL running on a node decides to change the node's place in the present version of its DODAG; causes the node to set its `prefpar` and `rank` to new values, potentially also modifying `minrank` to the new value of `rank`;
(d) *Neighbor entry addition*—occurs independently of RPL, for instance, when an external protocol for routing adjacency maintenance discovers a node's neighbor; causes a new entry, `n`, representing the neighbor to be added to the node's `neighborset`, with field `n.` *id* set to the neighbor's identifier, fields `n.` *version* and `n. rank` set to their initial values (as formalized in the next section), and the other fields, which are not controlled by RPL, set arbitrarily;
(e) *Neighbor entry removal*—also takes place independently of RPL, for example, when another protocol decides that a node's neighbor is no longer worth monitoring; causes the entry, `n`, representing the neighbor to be removed from the node's `neighborset`;
(f) *Neighbor entry update of non-RPL fields*—again, happens independently of RPL; for an existing entry, `n`, in the executing node's `neighborset`, can cause changes to any of the entry's fields except for `n. id`, `n. version`, and `n. rank`, which are controlled by RPL;
(g) *DIO message reception*—occurs when a packet containing a DIO message, `d`, in the delivery multiset, `linkdset`, for a link to a node from one of its neighbors is received and passed to RPL for processing; may cause a removal of message *d* from multiset *linkdset* (if this is the last instance of `d` to be received, as explained shortly); in addition, if an entry, `n`, corresponding to the neighbor (i.e. `n. id = d. id`) exists in the executing node's `neighborset`, causes fields `n.` *version* and `n. rank` of the entry to be set to the values of the corresponding fields, `d.` *version* and `d. rank`, of the message; otherwise, has no effect on the node's `neighborset`;
(h) *DIO message transmission*—takes place if RPL running on a node decides to transmit a packet with a DIO message, `d`, to a neighbor; causes `d` to be added to the delivery multiset, `linkdset`, for the link from the node to the neighbor, with field `d.` *id* equal to the node's identifier and fields `d.` *version* and `d.` *rank* equal to the node's variables `version` and `rank`, respectively.

Each event thus indeed represents a logically related group of program instructions that forms a certain whole. Likewise, each of them can in principle be executed at any time, if the executing node is live. This explains why node program counters are not necessary in the global system state.

Environment-driven events. The events triggered by the environment are in turn as follows:

(a) *DIO message loss*—occurs if a packet containing a DIO message, `d`, that is in transit over some link (i.e. is present in the link's delivery multiset, `linkdset`) is lost (i.e. will not be

received by the target node); causes `d` to be removed from `linkdset`;

(b) *Link start-up*—takes place when a dead link goes up; causes the link to become live;

(c) *Link death*—happens when a live link goes down; causes the link to become dead and may cause all or some of the messages in the link's delivery multiset, `linkdset`, to be removed from this multiset;

(d) *Node start-up*—occurs when a dead node (re)starts; causes the node to become live and sets its local variables to their initial values (following the rules formalized in the next section);

(e) *Node death*—takes place when a live node crashes; causes the node to become dead and makes the values of all its local variables undefined.

Whereas start-ups and crashes of particular nodes and links are typically defined per scenario, packet loss is an inherent feature of any low-power wireless network. For this reason, let us look more closely into modeling communication, especially since apart from packet loss, real-world communication may also exhibit packet corruption, duplication, and reordering, and, as emphasized previously, we strive to avoid any assumptions that would make our model unrealistic.

We start with packet corruption, which can manifest in a range of ways: from garbled bits to entire messages maliciously injected by an attacker. In all cases, the result is that a node receives a packet that has never been sent by any other node. As mentioned previously, routing protocols normally do not tolerate Byzantine failures and hence are not prepared to handle such spurious packets. Therefore, for protection, they—or, to be precise, entire network stacks they belong to—adopt a number of countermeasures at different layers: from checksums to various encryption schemes. In effect, core algorithms of the protocols typically assume that the problem of packet corruption is effectively dealt with elsewhere, which we formalize as the following property:

> *No creation*: If node B receives a packet over the link from node A, then the packet must have been earlier transmitted by node A over this link.

As the subsequent phenomena, let us consider packet loss and duplication as they both affect packet delivery guarantees. For the communication channel between the two processes in the example from Listing 1, we assumed *perfect delivery*, under which any transmitted packet is delivered exactly once. In contrast, assuming perfect delivery in low-power wireless networks is simply unrealistic, especially for broadcast transmissions that are heavily utilized by RPL for packets with DIO messages. In other words, we must not ignore the two phenomena in our model.

A major issue when formalizing non-zero loss and duplication is again that we want to make as minimal and realistic assumptions as possible. For instance, accepting a certain percentage of packet loss is not sensible, especially since some low-power wireless links may have a truly low and variable quality. Therefore, we take an opposite approach, assuming that both loss and duplication are unknown for any link and may vary arbitrarily in time, which we formalize as follows:

> *Finite loss*: If node A always eventually transmits a packet over a link to node B, then node B always eventually receives a packet over the link.
> *Finite duplication*: If node A eventually always does not transmit a given packet over a link to node B, then node B eventually always does not receive the packet over the link.

What the first property states is that if a node repeatedly forever transmits packets over a link, then the node on the other side of the link also repeatedly forever receives (some) packets over that link (recall the earlier explanation for the *always eventually* combination of temporal operators). There is no requirement as to which of the packets will actually be received or how many (consecutive) packets in such an infinite stream are allowed to be lost. In other words, there is no bound on packet loss and we cannot assume any specific chances of an individual packet being lost. However, thanks to being so extremely pessimistic, the property captures virtually all loss patterns encountered in the real world. Likewise, the second property expresses minimal assumptions on packet duplication. What it requires is only that the number of duplicates of a given packet be finite if the packet is transmitted a finite number of times, so that the reception of the packet's duplicates ceases at some point (recall the explanation for the *eventually always* combination of temporal operators). In other words, no node forever keeps receiving the same packet (unless the packet is transmitted *ad infinitum*). The two properties are thus indeed extremely weak, which makes any conclusions derived based on them readily applicable in the real world. They were inspired by what is sometimes referred to as *fair-loss delivery*[12] but the original definition of that delivery model differs, though. Moreover, for the sake of simplicity, their present formulation implicitly assumes that nodes A and B are always adjacent, that is, A, B, and the links between them are always live. We deal with this assumption when formalizing the actual axioms for our model.

As the final phenomenon, let us consider packet reordering. In principle, even packets transmitted over

the same link may arrive in any order, depending on the policies at the link layer. Consequently, we do not assume any particular packet delivery order, again being pessimistic.

Axioms describing RPL's operation

Given the definition of a global system state and the allowed transitions between such states, we are ready to formulate axioms describing when and how precisely specific transitions can take place. Considering the tutorial nature of this article, instead of the symbolic notation of LTL, we will continue using the natural language, like in the link properties from the previous section. In general, this is a common approach when analyzing distributed algorithms[12] as it greatly facilitates explaining the adopted reasoning and—when applied judiciously—need not make the reasoning "less formal." To support this claim, we illustrate that in particular the previous properties for links can be expressed in the symbolic notation of LTL.

To this end, let us define the following predicates for DIO message transmissions and receptions:

- $tx_dio(A, B, d)$ is true in any state $i>0$ of any computation σ iff the transition to this state corresponds to a transmission by node A of a given DIO message d over the link from A to B;
- $rx_dio(A, B, d)$ is true in any state $i>0$ of any computation σ iff the transition to this state corresponds to a reception by node B of a given DIO message d over the link from A to B;
- $tx_some(A, B)$ is true in any state $i>0$ of any computation σ iff the transition to this state corresponds to a transmission by node A of some DIO message over the link from A to B;
- $rx_some(A, B)$ is true in any state $i>0$ of any computation σ iff the transition to this state corresponds to a reception by node B of some DIO message over the link from A to B.

With these predicates, the *finite duplication* property can be translated literally for any computation σ, any nodes A and B, and any DIO message d:

Finite duplication:
$$\sigma \models \Diamond\Box\neg tx_dio(A, B, d) \rightarrow \Diamond\Box\neg rx_dio(A, B, d)$$

The same is true for *finite loss*:

Finite loss:
$$\sigma \models \Box\Diamond tx_some(A, B) \rightarrow \Box\Diamond rx_some(A, B)$$

Only the translation of the *no creation* property may arguably seem less straightforward:

No creation:
$$\sigma \models (\Box\neg rx_dio(A, B, d)) \vee (\neg rx_dio(A, B, d)\ \mathcal{U}\ (tx_dio(A, B, d) \wedge \neg rx_dio(A, B, d)))$$

The reason for such a seemingly involved translation is that there is no standard operator in the symbolic notation that would allow in a given state of a computation for referring to past states. Consequently, to ensure that a reception of a packet over a link is preceded by a transmission of this packet over the link, the formulation utilizes the temporal operator *until*, which enforces the desired order of events. More specifically, it states that a given DIO message, d, is either never received by node B over the link from A to B or is not received over this link until it has been transmitted by node A over the link. The last conjunction is necessary to ensure that transmission and reception of the same message do not happen simultaneously: the message can be received only after it has been transmitted and not also when it is being transmitted. Usually, with some practice, performing such translations into the symbolic notation is not overly difficult.

After this interlude, we can thus proceed to formulating the axioms describing RPL's operation, as derived from its specification. They are divided into four groups that correspond to distinct pieces of functionality necessary given our focus on RPL's DODAG formation and maintenance.

Control traffic axioms. The properties describing control traffic, that is, the way DIO messages are exchanged between neighboring nodes, can be formulated as follows.

CT1: *If a node receives a DIO message, `d`, then the message must have been transmitted earlier by the node's neighbor: `d`. id equals to the neighbor's identifier whereas d. version and d. rank equal, respectively, to the neighbor's version and rank from the moment of transmitting d.*

CT2: *If a node's neighbor is always eventually adjacent to the node, then the node always eventually receives a DIO message from this neighbor.*

CT3: *A node eventually never receives a given DIO message.*

They correspond directly to the previous link properties, albeit formulated in a manner that takes into account the way RPL utilizes DIO messages. More specifically, property CT1 corresponds to *no creation*, in addition defining how the values of fields `id`, `version`, and `rank`, are set in a message. CT2 is in turn equivalent to *finite loss* but, by taking into account adjacency, factors in node and link failures, and, by

considering that RPL transmits DIO messages perpetually, omits the satisfied transmission-related assumption from the original property. Finally, CT3 corresponds to *finite duplication*, again assuming that any DIO message is transmitted by a node finitely many times. All in all, the assumptions in the control traffic properties are extremely weak, which is to make any conclusions drawn under them broadly applicable in the real world.

Routing adjacency maintenance axioms. As the next group, let us formulate properties describing routing adjacency maintenance, that is, the rules for maintaining nodes' local `neighborsets`. On one hand, this functionality is by and large beyond RPL's specification: it is assumed to be performed by external solutions. On the other hand, to form and keep a DODAG, RPL relies on the nodes'`neighborsets` to be up to date. Formalizing the rules of routing adjacency maintenance is thus crucial to enable reasoning about RPL's behavior in a dynamic network.

Accordingly, let us start with two properties that, while not formulated explicitly in RPL's specification, can arguably be inferred from it.

RA1:*Always, if a node's* `neighborset` *changes (i.e. entries are added or removed, or their fields are modified), then the node eventually reselects its* `prefpar` *and* `rank`*, or dies.*

RA2:*Always, for each entry,* `n`*, in a node's* `neighborset`*,* `n.rank` *is infinite and* `n.version` *is equal to InitialDODAGVersion, if the node has received no DIO message,* `d`*, with* `d.id` *equal to* `n.id` *since the entry was added to the node's neighborset, or* `n.rank` *is equal to* `d.rank` *and* `n.version` *is equal to* `d.version`*, where d is the last DIO message with* `d.id` *equal to* `n.id` *received by the node since the entry was added to the node's* `neighborset`*.*

Property RA1 states that any change to a node's `neighborset` is followed by `prefpar` and `rank` reselection, unless the node dies. Note that not only is its formulation natural but also does not enforce any particular reselection policy: the reselection can be done immediately, after some time (e.g. in a different operating system context), or periodically. In other words, it does not constrain potential implementations beyond what is truly required.

Property RA2, in turn, formalizes consistency of those fields of `neighborset` entries that are fully controlled by RPL: their values for an entry for a given neighbor are copied from DIO messages from this neighbor. Again, this formalization can be inferred from RPL's specification.

In contrast, when it comes to formalizing the actual tracking of neighbors and their adjacency, the specification contains no precise information, leaving this issue to external solutions. Yet, to be able to prove anything, we do have to make some assumptions on the behavior of such solutions. We formalize these assumptions as properties RA3 and RA4.

RA3:*If a node's neighbor is eventually always nonadjacent, then an entry,* `n`*, with this neighbor's identifier as* `n.id` *eventually always either is absent from the node's* `neighborset` *or has its* `n.reachable` *flag set to* `false`*.*

RA4:*If a node's neighbor is eventually always adjacent, then an entry,* `n`*, with this neighbor's identifier as* `n.id` *is eventually always present in the node's* `neighborset` *and has its* `n.reachable` *flag set to* `true`*.*

Property RA3 describes the rules of detecting unreachable neighbors. To be considered unreachable, a node's neighbor, from some moment in time, has to permanently remain nonadjacent to the node (i.e. it has to be dead or its link with the node has to be down). This in particular means that neighbors that are temporarily nonadjacent need not be detected as unreachable—only ones that remain so for extended periods—in practice, depending on the timeouts of an actual failure detector. In other words, our assumptions on the failure detector are extremely weak, which makes any conclusions derived under them more broadly applicable in the real world. As a side note, our assumptions resemble those for what is known as *eventually perfect failure detector* in classic distributed algorithms.[12] Furthermore, regarding the marking of a neighbor entry as unreachable, the formulation of the property again does not impose any particular implementation: the marking can be done through the *reachable* flag or by removing the entry from the node's `neighborset`.

Property RA4 is symmetric to RA3 in that it considers reachable neighbors. Consequently, for brevity, we omit its discussion and proceed to formalizing another functionality.

DODAG versioning axioms. The next group of properties describes the management of DODAG versions. Unlike the previous one, this functionality does belong to RPL. Nevertheless, some of its aspects, notably those related to liveness, are still underspecified. Therefore, like previously, let us start with those properties that can arguably be inferred from RPL's specification.

DV1:*When a node starts for the first time, its version is set to* `InitialDODAGVersion`*.*

DV2: *Always, if a node changes the value of its version (i.e. either generates or adopts a value), then it eventually reselects its* `prefpar` *and* `rank`*, or dies.*

DV3: *Always, a node's* `version` *does not decrease.*

Property DV1 defines the initial value of a node's `version` variable, thereby not requiring any further comment.

Property DV2 entails `prefpar` and `rank` reselection after any change to a node's `version` variable. It thus expresses the very goal of introducing a new version for a DODAG, that is, forcing the DODAG to be rebuilt. Like RA1 in the case of `neighborset` changes, DV2 does not specify precisely when such a reselection should take place, thereby not constraining implementations.

Finally, property DV3 states that DODAG versions are monotonic. It is worth mentioning that while formalizing the intent of RPL's designers, this property is only approximated but not guaranteed by the protocol. First, because of a limited width (8 bits) of `version` variables in RPL's specification, they can overflow. Second, since they are not backed up to stable storage (e.g. flash), they are reset to their initial values after a node's restart. The effect in both cases is that DV3 can be violated in the real world. It can be shown formally that such violations may preclude nodes from installing a new DODAG version. Consequently, to minimize the risk of such situations, rather than plain 8-bit integers for `version`s, RPL's specification adopts so-called lollipop counters.[59] Moreover, even if such a situation occurs, it can be recovered from by letting some DODAG version converge and then introducing a new one. For these reasons, we assume property DV3 to be true.

The next aspect concerns DODAG version generation. By and large, RPL's specification leaves this issue open to implementers, notably when it comes to which node introduces a new version and when. In principle, we could assume that any node is allowed to start a new DODAG version. In practice, however, for management reasons, it is typically the root node that generates new versions, while the other nodes adopt them based on information from their `neighborset`s. Therefore, being able to follow either of the two approaches, we choose the latter one here. In contrast, we defer specifying when a new version is generated to particular operational scenarios, if they need this. Overall, the assumptions on generating DODAG versions are formalized as property DV4.

DV4: *Always, if a node changes its* `version` *to some value,* `v`*, then either the node is the root and* `v` *is a new value generated by it, or the node has an entry,* `n`*, in its* `neighborset` *such that* `n.version` *equals to* `v`*, and hence* `v` *is an adopted value.*

Related to DODAG version generation is adoption of generated versions, which is also the last aspect in the considered group of properties. Like previously, RPL's specification contains virtually no requirements on when a node should adopt a new DODAG version. However, never demanding the node to do so precludes liveness: a DODAG version newly generated by the root node may never be adopted by any other node. On the other hand, aggressively forcing a node to adopt any new version it learns about may be an overkill. In search for a middleground, we thus oblige a node to change its `version` to a newer one only if not doing so would make it always lag behind some of its neighbors. Moreover, we do not force the node to change its `version` to a particular value: it may select the adopted version on its own. These weak requirements are formalized as property DV5, which completes the group of axioms related to DODAG versions management.

DV5: *Always, if a node's* `version` *equals to some value,* `v`*, and there always exists in the node's* `neighborset` *an entry,* `n`*, such that* `n.version` *is greater than* `v` *and* `n.reachable` *is* `true`*, then eventually the node's* `version` *is not equal to* `v`*.*

Objective function axioms. As the last group, we consider axioms describing the selection of a node's preferred parent and rank in a DODAG version. As mentioned previously, this functionality is delegated to objective functions, such as OF0[55] and MRHOF.[56] In principle, they are treated by RPL as "black boxes." Nevertheless, the protocol, sometimes implicitly, does define a few requirements on their results, which we gather into the following properties.

OF1: *A node's* `prefpar` *and* `rank` *change only as a result of reselection, a change of the node's* `version`*, or the node's death and (re)start; otherwise, they remain unmodified.*

OF2: *A node's* `minrank` *is always equal to the minimal value of the node's* `rank` *either since the node last changed its* `version` *or since the node has first started if it has never changed its* `version` *from the* `InitialDODAGVersion` *value.*

OF3: *Always, when reselecting its* `prefpar` *and* `rank`*, the root node adopts null and* `MinHopRankIncrease`*, respectively. These are also the initial values of the two variables at the root node when the node (re)starts or changes its version.*

OF4: *Always, when reselecting its* `prefpar` *and* `rank`*, a non-root node adopts* `null` *as* `prefpar` *iff it also adopts infinite* `rank`*. These are also the initial values of the two variables at the non-root node when the node (re)starts or changes its* `version`*.*

OF5: *Always, when reselecting its* `prefpar` *and* `rank`*, a non-root node adopts* `null` *and infinity, respectively, iff its* `neighborset` *does not contain an entry,* n*, for which a potential rank,* r*, can be computed (in an objective-function-specific way), such that* n *and* r *satisfy all the following constraints*:

(a) $n.\texttt{version} = \texttt{version}$,
(b) $n.\texttt{rank} < \textit{infinity}$,
(c) $n.\texttt{reachable} = \texttt{true}$,
(d) $r < \textit{infinity}$,
(e) $r \geqslant n.\texttt{rank} + \texttt{MinHopRankIncrease}$,
(f) $r \leqslant \texttt{minrank} + \texttt{MaxVersionRankIncrease}$.

Otherwise, the node adopts $n.\texttt{id}$ and r as its `prefpar` and `rank`, respectively, for some neighbor n, such that n and r satisfy all conditions (a)–(f).

Property OF1 defines the only events that affect a node's `prefpar` and `rank` variables.

Property OF2 formalizes the dependency between a node's `rank`, `minrank`, and `version`. Again, it is worth mentioning that it reflects the intent of RPL's designers and is only approximated by the protocol. This is because, like a node's other variables, `minrank` is not required to be backed up to stable storage, and hence if the node reboots, the value of this variable need not be restored. In contrast, if we dropped OF2 for node crashes and restarts, it can be formally shown that RPL would not be able to recover from some failures, such as network partitions. Such problematic scenarios are rather unlikely, though. Moreover, nothing prevents RPL's implementation from backing up `minrank`. Therefore, we assume OF2 to hold also in the presence of node crashes and reboots.

Property OF3 specifies that the root node's `rank` in any DODAG version is fixed, equal to `MinHopRankIncrease`, and, regarding its position in a DODAG, the node never has any `prefpar`.

Finally, properties OF4 and OF5 are similar in that they describe rules for `prefpar` and `rank` selection by a non-root node. Property OF4 entails that a non-root node without a `prefpar` has to adopt an infinite `rank` and vice versa, which represents the fact that it does not have a routing path to the root node. Property OF5, in turn, gives conditions for a non-root node's neighbor, n, under which the neighbor can be considered as the node's `prefpar`: condition (a) states that, at least given the node's knowledge, the neighbor has to belong to the same DODAG version; condition (b)—that the neighbor must not have an infinite rank, at least it must have advertised a finite rank to the node; condition (c)—that the neighbor has to be considered reachable by the node; condition (d)—that the rank the node would have if this neighbor were its preferred parent, a so-called *potential* `rank`, r, has to be finite; condition (e)—that the potential rank has to be greater from the neighbor's rank at least by a constant, `MinHopRankIncrease`; and condition (f)—that the potential rank must not exceed the node's `minrank` by more than another constant, `MaxVersionRankIncrease`. All these conditions can be inferred from RPL's specification. The operation of an objective function, in turn, is modeled in property OF5 in two ways. First, the property does not specify how the potential ranks are computed, which is left to the particular objective function employed. Second, the property does not specify which of the neighbors satisfying all conditions (a)–(f) is selected as a node's preferred parent. In other words, under property OF5, `prefpar` and *rank* are indeed selected by the objective function, which appears as a "black box" to RPL, while they are guaranteed to satisfy the conditions necessitated by the protocol.

Summary of lessons learned

All in all, the presented model does cover the aspects fundamental to studying the dynamic behavior of RPL's DODAG construction and maintenance. Its system state includes all the elements that contribute to a DODAG at a particular moment: nodes, links, messages, versions, ranks, preferred parents, neighbor tables, adjacency, and routing metric values. At the same time, the state exhibits features characteristic to distributed systems, such as distribution of information among nodes and messages in transit, knowledge inconsistency between different nodes and even at a single node, communication with loss, duplication, and out of order delivery, and node and link failures and recoveries, to name a few prominent examples. The transitions between such states are also granular enough to model the evolution of these features in time. Finally, the rules for the transitions have been meticulously inferred from protocol descriptions, implementations, and community knowledge, so that the dynamic behavior they model comes close to what is observed in a real-world system, which we confirmed, among others, empirically.[6–9]

When presenting these issues, we aimed to illustrate typical problems that have to be faced when devising a formal model of a routing protocol. We highlighted the need for limiting complexity by focusing on features that are vital to the phenomena of interest. We showed how to model processing and communication, abstract algorithms comprising the protocol, and approach open issues and external solutions on which the protocol depends. Throughout our discussion, we emphasized the need for avoiding oversimplifying and overspecifying the model, which would otherwise make its behavior deviate from that observable in the real world, thereby limiting its potential for deriving practically relevant conclusions. Therefore, even though parts of the model can likely be reused out of the box for other routing protocols, we believe that it is the knowledge

we aimed to share when presenting it that can guide the readers in their own modeling attempts.

What we have not discussed yet is in turn the iterative nature of a typical modeling process. In particular, the presented model is a result of over a dozen iterations that improved its various aspects. We illustrate how such adjustments can be performed at the end of the next section, as this requires some practice in verification.

Verifying hypotheses on RPL's behavior

Given a model that describes the dynamic behavior of a routing protocol, we can formulate hypotheses regarding this behavior in various situations. We can then employ the LTL reasoning rules discussed previously to try to prove that a particular hypothesis holds for the model.

One possible outcome of such a verification attempt is a counterexample that identifies a specific scenario in which the hypothesis is violated. Analyzing such a counterexample we can conclude that the hypothesis is indeed false. In particular, many hypotheses that we formulated for RPL and that seemed intuitive at first turned out not to hold in the end. In effect, we were forced to reformulate them or abandon altogether, in both cases gaining new insights.

However, it may also be the case that we are missing some assumptions in our model or in the particular scenario the hypothesis considers. Armed with this knowledge, we may revise the model or make the considered scenario more specific. This reinforces our previous remark that a protocol verification process is typically iterative, alternating between modeling the protocol and (dis)proving hypotheses on its behavior, which is how our models of RPL emerged.

In any case, a product of the process that is at least equally important as a proof of a hypothesis is precise information on what properties of the model and the considered scenario are crucial for the hypothesis to hold. This knowledge can be utilized in practice at virtually all stages of protocol development. At the design stage, it can determine the architecture of the algorithms constituting the protocol and can drive their specification: one way or the other the information has to be put in the specification to facilitate building correct implementations. At the implementation and testing stages, it can help develop correct code: having the crucial properties explicitly formulated, it is much easier to maintain them in the code and to devise dedicated test scenarios. Finally, at the deployment stage, being aware of the assumptions on the scenarios in which the hypothesis holds gives more confidence in the reliability of the protocol in the target environment. In particular, in the case of RPL, even though the specification and implementations had already existed for several years, the aforementioned findings from our modeling and verification process still turned out relevant in practice,[6,7] as we summarize further in the paper.

To this end, however, apart from an appropriate methodology and models, which were covered in the previous sections, one also requires suitable proof techniques. In particular, as mentioned previously, even if some hypotheses can be verified through (semi-)automated model checkers, manual proof techniques are typically necessary to generalize the results to other network configurations, protocol parameter settings, and the like. In this section, we thus give examples of the main techniques that we adopted and developed when proving various hypotheses for RPL. The techniques are discussed with selected hypotheses serving as running examples. Considering the tutorial nature of this article, the choice of the hypotheses aims to be illustrative rather than exhaustive. Nevertheless, we cover both safety and liveness properties.

Proving safety properties

Safety properties are often formulated as invariants that have to hold in all or in well-defined states of computations. A fundamental technique for proving an invariant in LTL is mathematical induction on time, that is, on the sequence of states representing a computation. This is precisely what we did when proving Formula (2) for the example from Listing 1. To conduct such a proof (cf. Figure 5), we take an arbitrary computation that is possible in the scenario that the property considers. Then, as the inductive base, we need to prove that the property holds in some state of the computation, typically but not necessarily the first state. Finally, as the inductive step, from the fact that the property holds in an arbitrary state (and all previous states) of the computation, we have to derive that it also holds in the next state. Depending on the property, rather than all states, we may consider only those that satisfy some specific criteria.

Sample property. To illustrate the use of this technique, we will show the lower bound on ranks in RPL, that is, the fact that all ranks ever appearing in the system are

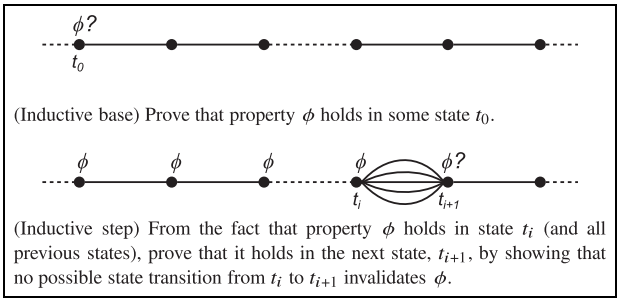

Figure 5. A typical approach of verifying an invariant by induction on time in an arbitrary computation.

at least `MinHopRankIncrease`. More specifically, we will prove the following lemma.

Lemma 1. Always, if a node is live, its variable rank and fields `rank` of all entries in its `neighborset` are greater than or equal to `MinHopRankIncrease`. Likewise, always, field `rank` of any DIO message in transit is greater than or equal to `MinHopRankIncrease`.

Note that we named the property as "lemma" instead of "hypothesis." This is to indicate that it is true. Irrespective of the naming convention, however, the property itself is not mentioned explicitly in RPL's specification. Nevertheless, as we will demonstrate, it can be derived from our model of the protocol and hence from the specification itself. Having such an explicit formulation facilitates adding checks for the property in the code of RPL's implementations. Considering that `MinHopRankIncrease` is greater than zero, this can, for instance, help protect the protocol from rank corruption errors that once led to a collapse of ARPANET routing.[60]

Before delving into the proof, however, let us introduce some notation that will make our reasoning more succinct. More specifically, we define the following multisets:

- $Ranks_{\text{Vars}}(t) = \bigcup_{X \in \text{LiveNodes}(t)} \{ \texttt{rank}_X(t) \}$, which encompasses the values in state t of variable `rank` for all nodes that are live in this state;
- $Ranks_{\text{NbrEntrs}}(t) = \bigcup_{X \in \text{LiveNodes}(t)} \{ n.\texttt{rank}(t) \mid n \in \texttt{neighborset}_X(t) \}$, which contains the values in state t of fields $n.$ `rank` of all entries, n, in the `neighborset` s of all nodes that are live in state t;
- $Ranks_{\text{DIOs}}(t) = \bigcup_{\langle Y,X \rangle \in \text{Links}} \{ d.\texttt{rank}(t) \mid d \in \texttt{linkdset}_{\langle Y,X \rangle}(t) \}$, which comprises the values in state t of fields $d.$ `rank` of all DIO messages, d, that are in transit in state t.

We will denote the union of these multisets as $Ranks_*(t) = Ranks_{\text{Vars}}(t) \cup Ranks_{\text{NbrEntrs}}(t) \cup Ranks_{\text{DIOs}}(t)$. Note that they are all multisets rather than plain sets because a given value may appear in a particular multiset many times. For example, many nodes can simultaneously have their ranks infinite, and we would like to keep track of each such rank. Moreover, careful readers may have noticed that for $Ranks_{\text{Vars}}$ and $Ranks_{\text{NbrEntrs}}$, we consider only nodes that are live in a given state, whereas for $Ranks_{\text{DIOs}}$, we consider all links. The same is true in the case of the lemma itself. The reason is that the values of variables `rank` and `neighborset` are undefined if a node is dead. In contrast, a delivery multiset, `linkdset`, is defined for any link (i.e. either live or dead) to account for the various places a message in transit may be at, as explained previously.

With this notation, Lemma 1 can be reformulated as follows:

For any state $t \in \{0, 1, 2, \ldots\}$ of any computation involving our model, for all $r \in Ranks_*(t)$, we have $r \geqslant$ `MinHopRankIncrease`.

Inductive proof. We are now ready of give a proof of the lemma.

Proof. Consider an arbitrary computation for our model.

The inductive base is the first state: $t = 0$. In this state, there are no DIO messages in transit yet because there is no previous state transiting from which a node could have transmitted a DIO, and DIOs do not appear spontaneously (property CT1). In other words, $Ranks_{\text{DIOs}}(0) = \emptyset$, and hence for any $r \in Ranks_{\text{DIOs}}(0)$, we have $r \geqslant$ `MinHopRankIncrease`. When it comes to nodes, in turn, they are all dead in state 0. Therefore, $Ranks_{\text{Vars}}(0) = \emptyset$ and $Ranks_{\text{NbrEntrs}}(0) = \emptyset$, and hence again for any $r \in Ranks_{\text{Vars}}(0) \cup Ranks_{\text{NbrEntrs}}(0)$, we have $r \geqslant$ `MinHopRankIncrease`. All ranks in the system are thus indeed at least `MinHopRankIncrease` in state 0 of the computation, that is, for all $r \in Ranks_*(0)$, we have $r \geqslant$ `MinHopRankIncrease`, which constitutes the inductive base.

For the inductive step, we take an arbitrary state, $t \geqslant 0$, and assume that in this state all ranks appearing in the system are at least `MinHopRankIncrease`, that is, for all $r \in Ranks_*(t)$, we have $r \geqslant$ `MinHopRankIncrease`. We will show that no event in the system generates a rank value smaller than `MinHopRankIncrease`, and thus in the next state, $t + 1$, of the computation, for all $r' \in Ranks_*(t + 1)$, we also have $r' \geqslant$ `MinHopRankIncrease`.

To this end, we analyze what effects each possible event has on $Ranks_*(t + 1)$ if this event corresponds to the transition of the system from state t to state $t + 1$.

(a) *Node start-up.* If the event corresponding to the transition of the system from state t to state $t + 1$ is a start of some node X, the following dependencies hold: $Ranks_{\text{Vars}}(t + 1) = Ranks_{\text{Vars}}(t) \cup \{ \texttt{rank}_X(t + 1) \}$, $Ranks_{\text{NbrEntrs}}(t + 1) = Ranks_{\text{NbrEntrs}}(t) \cup \{ n.\texttt{rank}(t + 1) \mid n \in \texttt{neighborset}_X(t + 1) \}$, and $Ranks_{\text{DIOs}}(t + 1) = Ranks_{\text{DIOs}}(t)$. In other words, X's `rank` and fields `rank` of all entries in X's `neighborset` appear in the appropriate multisets, whereas the multiset of DIOs in transit remains unchanged. If X is the root node, then $\texttt{rank}_X(t + 1) =$ `MinHopRankIncrease`

(property OF3); otherwise, $\texttt{rank}_X(t+1) = \infty$ (property OF4). In both cases, we thus have $\texttt{rank}_X(t+1) \geqslant \texttt{MinHopRankIncrease}$. Consequently, since for all $r \in Ranks_{\text{Vars}}(t)$, $r \geqslant \texttt{MinHopRankIncrease}$, also for all $r' \in Ranks_{\text{Vars}}(t+1)$, $r' \geqslant \texttt{MinHopRankIncrease}$. Similarly, for each entry, n, in X's $\texttt{neighborset}$ in state $t+1$, we have $n.\texttt{rank}(t+1) = \infty$ (property RA2). Therefore, since for all $r \in Ranks_{\text{NbrEntrs}}(t)$, $r \geqslant \texttt{MinHopRankIncrease}$, also for all $r' \in Ranks_{\text{NbrEntrs}}(t+1)$, $r' \geqslant \texttt{MinHopRankIncrease}$. Finally, since $Ranks_{\text{DIOs}}(t+1) = Ranks_{\text{DIOs}}(t)$ and for all $r \in Ranks_{\text{DIOs}}(t)$, $r \geqslant \texttt{MinHopRankIncrease}$, also for all $r' \in Ranks_{\text{DIOs}}(t+1)$, $r' \geqslant \texttt{MinHopRankIncrease}$. All in all, for all $r' \in Ranks_*(t+1)$, we have $r' \geqslant \texttt{MinHopRankIncrease}$.

(b) *Node death.* If the event corresponding to the transition from state t to $t+1$ is a death of some node X, then the multisets do not get any new elements; on the contrary, they can shrink. To be precise, the following dependencies hold: $Ranks_{\text{Vars}}(t+1) \subset Ranks_{\text{Vars}}(t)$, $Ranks_{\text{NbrEntrs}}(t+1) \subseteq Ranks_{\text{NbrEntrs}}(t)$, and $Ranks_{\text{DIOs}}(t+1) \subseteq Ranks_{\text{DIOs}}(t)$. Therefore, since $Ranks_*(t+1) \subset Ranks_*(t)$ and for all $r \in Ranks_*(t)$, $r \geqslant \texttt{MinHopRankIncrease}$, we can conclude that for all $r' \in Ranks_*(t+1)$, $r' \geqslant \texttt{MinHopRankIncrease}$.

(c) *Link start-up.* A start-up of some link does not affect any of the multisets, and hence for all $r' \in Ranks_*(t+1) = Ranks_*(t)$, $r' \geqslant \texttt{MinHopRankIncrease}$.

(d) *Link death.* Upon a death of some link $\langle X, Y\rangle$, only $Ranks_{\text{DIOs}}$ is affected because some DIO messages in transit between node X and node Y may get lost, that it, $Ranks_{\text{Vars}}(t+1) = Ranks_{\text{Vars}}(t)$, $Ranks_{\text{NbrEntrs}}(t+1) = Ranks_{\text{NbrEntrs}}(t)$, and $Ranks_{\text{DIOs}}(t+1) \subseteq Ranks_{\text{DIOs}}(t)$. Consequently, for all $r' \in Ranks_*(t+1) \subseteq Ranks_*(t)$, we have $r' \geqslant \texttt{MinHopRankIncrease}$.

(e) *Neighbor entry addition.* If the event corresponding to the transition from state t to state $t+1$ is an addition of a previously nonexistent entry, n, to node X's $\texttt{neighborset}$, the value of the entry's field $\texttt{rank}$, which is equal to infinity (property RA2), appears in $Ranks_{\text{NbrEntrs}}(t+1)$. In effect, for all $r' \in Ranks_*(t+1) = Ranks_*(t) \cup \{\infty\}$, we have $r' \geqslant \texttt{MinHopRankIncrease}$.

(f) *Neighbor entry update of non-RPL fields.* This event does not affect any of the multisets, and thus for all $r' \in Ranks_*(t+1) = Ranks_*(t)$, $r' \geqslant \texttt{MinHopRankIncrease}$.

(g) *Neighbor entry removal.* If the event during the transition from state t to $t+1$ is a removal of a previously existing entry, n, from some X's $\texttt{neighborset}$, the corresponding multiset is shrunk by the value of $n.\texttt{rank}(t)$. As a result, the following dependencies hold $Ranks_{\text{Vars}}(t+1) = Ranks_{\text{Vars}}(t)$, $Ranks_{\text{NbrEntrs}}(t+1) \subset Ranks_{\text{NbrEntrs}}(t)$, and $Ranks_{\text{DIOs}}(t+1) = Ranks_{\text{DIOs}}(t)$, and hence for all $r' \in Ranks_*(t+1) \subset Ranks_*(t)$, we must have $r' \geqslant \texttt{MinHopRankIncrease}$.

(h) *DIO message transmission.* If the event causing the transition from state t to state $t+1$ is a transmission of a DIO message, d, by some node X to node Y, then d is added to *linkdset* of link $\langle X, Y\rangle$. Field $\texttt{rank}$ of the message is copied from X's variable *rank* at the time of transmission, that is, $d.\texttt{rank}(t+1) = \texttt{rank}_X(t)$ (property CT1). If effect, $Ranks_{\text{DIOs}}(t+1) = Ranks_{\text{DIOs}}(t) \cup \{\texttt{rank}_X(t)\}$, while the other multisets remain unmodified: $Ranks_{\text{Vars}}(t+1) = Ranks_{\text{Vars}}(t)$ and $Ranks_{\text{NbrEntrs}}(t+1) = Ranks_{\text{NbrEntrs}}(t)$. However, since $\texttt{rank}_X(t) \in Ranks_{\text{Vars}}(t) \subseteq Ranks_*(t)$ and for all $r \in Ranks_*(t)$, $r \geqslant \textit{MinHopRankIncrease}$, we know that $\texttt{rank}_X(t) \geqslant \texttt{MinHopRankIncrease}$. Taking everything into account, we can thus conclude that for all $r' \in Ranks_*(t+1)$, we have $r' \geqslant \texttt{MinHopRankIncrease}$.

(i) *DIO message reception.* If the event is in turn a reception by node X of a DIO message, d, from node Y, then two multiset changes may result. First, d may be removed from the *linkdset* for link $\langle Y, X\rangle$, if this is the last reception of the message (i.e. X will receive no more duplicates of d over link $\langle Y, X\rangle$). In other words, $Ranks_{\text{DIOs}}(t+1) \subseteq Ranks_{\text{DIOs}}(t)$, and hence for all $r' \in Ranks_{\text{DIOs}}(t+1)$, we have $r' \geqslant \texttt{MinHopRankIncrease}$. Second, the entry, n, that corresponds in X"s $\texttt{neighborset}$ to node Y (i.e. $n.\texttt{id}(t+1) = d.\texttt{id}(t) = Y$) is updated such that $n.\texttt{rank}(t+1) = d.\texttt{rank}(t)$ (property RA2). In other words, $Ranks_{\text{NbrEntrs}}(t+1) = Ranks_{\text{NbrEntrs}}(t) \setminus \{n.\texttt{rank}(t)\} \cup \{d.\texttt{rank}(t)\}$. Yet, $d.\texttt{rank}(t) \in Ranks_{\text{DIOs}}(t) \subseteq Ranks_*(t)$ and for all $r \in Ranks_*(t)$, we have $r \geqslant \texttt{MinHopRankIncrease}$. Therefore, we also have $d.\texttt{rank}(t) \geqslant \texttt{MinHopRankIncrease}$, and hence for all $r' \in Ranks_{\text{NbrEntrs}}(t+1)$, $r' \geqslant \texttt{MinHopRankIncrease}$. Finally, $Ranks_{\text{Vars}}(t+1) = Ranks_{\text{Vars}}(t)$ because the event does not affect the nodes" variable $\texttt{rank}$. We can thus conclude that for all $r' \in Ranks_*(t+1)$, indeed $r' \geqslant \texttt{MinHopRankIncrease}$.

(j) *DIO message loss.* In the case of a DIO message loss, a DIO message is removed from *linkdset* of some link $\langle X, Y \rangle$, and hence $Ranks_{\mathrm{DIOs}}(t+1) \subset Ranks_{\mathrm{DIOs}}(t)$ while $Ranks_{\mathrm{Vars}}(t+1) = Ranks_{\mathrm{Vars}}(t)$ and $Ranks_{\mathrm{NbrEntrs}}(t+1) = Ranks_{\mathrm{NbrEntrs}}(t)$. Consequently, for all $r' \in Ranks_*(t+1) \subset Ranks_*(t)$, we have $r' \geqslant$ `MinHopRankIncrease`.

(k) *Parent and rank reselection.* If the event corresponding to the transition from state t to state $t+1$ is a parent and rank reselection at some node X, then $Ranks_{\mathrm{Vars}}(t+1) = Ranks_{\mathrm{Vars}}(t) \setminus \{$ `rank` $_X(t)\} \cup \{$ `rank` $_X(t+1)\}$ while $Ranks_{\mathrm{NbrEntrs}}(t+1) = Ranks_{\mathrm{NbrEntrs}}(t)$ and $Ranks_{\mathrm{DIOs}}(t+1) = Ranks_{\mathrm{DIOs}}(t)$. Therefore, to ensure that for all $r' \in Ranks_*(t+1)$,$r' \geqslant$ `MinHopRankIncrease`, we have to show that `rank` $_X(t+1) \geqslant$ `MinHopRankIncrease`. There are three possible cases that dictate the value of `rank` $_X(t+1)$. First, if X is the root node, then `rank` $_X(t+1) =$ `MinHopRankIncrease` (property OF3). Second, if X is not the root and is unable to select its `prefpar`, then `rank` $_X(t+1) = \infty$ (property OF4). Third, if X is not the root but manages to select its `prefpar`, then `rank` $_X(t+1) = r_X$ for some value $r_X < \infty$ (property OF5). However, we must have $r_X \geqslant$ `n.rank` $(t) +$ `MinHopRankIncrease` for some entry `n` in X's `neighborset` (property OF5 5). Since `n.rank` $(t) \in Ranks_{\mathrm{NbrEntrs}}(t) \subset Ranks_*(t)$ and for all $r \in Ranks_*(t)$, $r \geqslant$ `MinHopRankIncrease`, we must have $r_X \geqslant 2 \times$ `MinHopRankIncrease`. In other words, in all three cases, `rank` $_X(t+1) \geqslant$ `MinHopRankIncrease`, and hence we can conclude that for all $r' \in Ranks_*(t+1)$, we have $r' \geqslant$ `MinHopRankIncrease`.

(l) *DODAG version change (i.e. generation or adoption).* The same dependency between the multisets occurs for a DODAG version change at some node X: $Ranks_{\mathrm{Vars}}(t+1) = Ranks_{\mathrm{Vars}}(t) \setminus \{$ `rank` $_X(t)\} \cup \{$ `rank` $_X(t+1)\}$, $Ranks_{\mathrm{NbrEntrs}}(t+1) = Ranks_{\mathrm{NbrEntrs}}(t)$, and $Ranks_{\mathrm{DIOs}}(t+1) = Ranks_{\mathrm{DIOs}}(t)$. This time, however, `rank` $_X(t+1)$ can attain two values: if X is the root node, then `rank` $_X(t+1) =$ `MinHopRankIncrease` (property OF3); otherwise, `rank` $_X(t+1) = \infty$ (property OF4). In both cases, we can conclude that for all $r' \in Ranks_*(t+1)$, we have $r' \geqslant$ `MinHopRankIncrease`. This completes the inductive step.

The proofs of the inductive base and the inductive step together confirm that in any state $t \in \{0, 1, 2, \ldots\}$ of the selected computation, for all $r \in Ranks_*(t)$, we have $r \geqslant$ `MinHopRankIncrease`. Since the computation has been chosen arbitrarily among all possible computations for our model, the lemma holds in any such computation, which ends the proof. □

Further examples of safety properties. To reinforce the claim that induction on the sequence of states constituting a computation is indispensable for proving safety properties, we give Lemmas 2 and 3, leaving their proofs with our model as an exercise for the interested readers.

Lemma 2. Always, if a node is live, its variable `rank` either is infinite or does not exceed its `minrank + MaxVersionRankIncrease`.

Lemma 3. In any state, let v_{root} be the value of the root node's variable `version`, if the root node is live in this state, or else, if the root node is dead, either the last value of variable `version` that the root node had before it died or `InitialDODAGVersion` if it has never started up. With this notation, always, if a node is live, its variable `version` and fields `version` of all entries in its `neighborset` are less than or equal to v_{root}. Likewise, always, field `version` of any DIO message in transit is less than or equal to v_{root}.

Lemma 2 is complementary to Lemma 1 in that it gives an upper bound on a node's rank at any time. Its proof is simpler than the proof of Lemma 1 because fewer events directly affect a node's variable `rank`. As a side note, in a few cases, the aforementioned incorrect behavior that we demonstrated for the two popular RPL's open-source implementations under failures[6,9] was correlated with the violation of Lemma 2, resulting from bugs that prevented the implementations from satisfying some of the properties of our model that are crucial for the lemma to hold.

Lemma 3, in turn, bounds the DODAG version a node may have. Its proof is very similar to the presented one but for each event involving a DODAG version, we have to consider two cases: one in which the event affects the root node and the other in which a non-root node is affected. We will utilize the lemma in another proof in the next section.

Further examples of safety properties can also be found in our previous papers.[6,7]

Proving liveness properties

While proving safety often boils down to what we presented hitherto, proving liveness may be more intricate, as a particular technique may be strongly dependent on what precisely is being proved. The techniques that we

will present in this and subsequent sections illustrate this claim, as they have been developed for proving liveness with regard to particular aspects of RPL. Since distance-vector routing protocols in general employ similar solutions, (some of) the techniques, possibly after some adaptation, may likely be reused also for other protocols. Nevertheless, by no means do we consider the techniques a complete repertoire for proving liveness. On the contrary, we envision that their prospective users will encounter problems that will require novel approaches. Therefore, our main goal is to discuss carefully selected examples so as to provide the readers with some directions and inspiration for attacking such problems.

We start with local liveness properties, normally involving a single node or a couple of nodes. We then exemplify proving global properties, referring to the entire node population.

Example of a local liveness property. Liveness properties that involve a small number of nodes can often be derived from the properties comprising the model without any sophisticated techniques. Occasionally, even (semi-)automated model checkers can help to this end. More advanced examples, in turn, include proofs by contradiction or some form of induction. Since we will illustrate the use of contradiction in a proof of a global liveness property in the next section, in this section we discuss an example of custom induction for proving a local liveness property, formulated as Lemma 4. As a side note, we will use this lemma to prove the aforementioned global liveness property.

Lemma 4. Always, if a node's `version` is equal to some value, v_{low}, and there always exists in the node's `neighborset` an entry, `n`, such that `n. version` is greater than or equal to some $v_{high} \geq v_{low}$ and `n. reachable` is `true`, then eventually always the node's `version` will be no less than v_{high}.

In short, the lemma combines the properties describing the adoption of new DODAG versions, notably properties DV5 and DV3, into something useful: the premises of the lemma are the same as of property DV5 but the conclusions are stronger. It states that if a node is aware of a DODAG version that is newer than its own one and that has already been adopted by a stable neighbor (i.e. one with which the node has a reliable link for a sufficiently long time), then eventually the node will abandon its current DODAG version and join a new one: either the one adopted by the neighbor or some yet newer. The difficulty in proving the lemma is that nothing in the model (and in RPL's specification) forces a node to adopt a *specific* DODAG version and hence it can potentially change its version multiple times. Worse yet, so can its neighbors in the meantime. We will use induction to show that irrespective of the number of such changes that may be necessary, the node's DODAG version will forever reach or exceed the given neighbor's version.

Proof. Consider an arbitrary computation for our model, an arbitrary node X participating in this computation, and an arbitrary state of the computation, t_s, in which the premises of the lemma hold, that is, for some v_{low} and v_{high} such that $v_{high} \geq v_{low}$, we have: $\texttt{version}_X(t_s) = v_{low}$ and for every state $t \geq t_s$, there exists $n \in \texttt{neighborset}_X(t)$, such that $\texttt{n. version}(t) \geq v_{high}$ and $\texttt{n. reachable}(t) = \texttt{true}$ (`n` can vary between the various states t). To prove the lemma, we need to show that starting from some state $t_f \geq t_s$, X's `version` will forever be no less than v_{high}, that is, for all $t' \geq t_f$, we will have $\texttt{version}_X(t') \geq v_{high}$.

To conduct this proof, we will use induction on the difference between v_{high} and v_{low}, that is, on $v_\Delta = v_{high} - v_{low}$. We can conduct such a proof because v_Δ belongs to nonnegative integers, for which a natural order exists: each number has well defined predecessors.

As the inductive base, let us take $v_\Delta = 0$. In this case, already in state t_s, we have $\texttt{version}_X(t_s) = v_{high} = v_{low}$. Moreover, since X's `version` never decreases (property DV3), for all $t' \geq t_s$, we have $\texttt{version}_X(t') \geq v_{high}$. Consequently, we can take $t_f = t_s$, which establishes the inductive base.

For the inductive step, in turn, we assume that the lemma holds for all $v_\Delta \leq i$, for an arbitrary $i \geq 0$. We will show that it also holds for $v_\Delta = i + 1 > 0$.

More specifically, from the premises of the lemma, we know that in state t_s, $\texttt{version}_X(t_s) = v_{low}$ and for every state $t \geq t_s$, there exists $n \in \texttt{neighborset}_X(t)$, such that $\texttt{n. version}(t) \geq v_{high}$ and $\texttt{n. reachable}(t) = \texttt{true}$. Since $v_\Delta > 0$, we have $v_{high} = v_{low} + v_\Delta > v_{low}$, and hence can conclude that there exists a state, $t_m > t_s$, such that $\texttt{version}_X(t_m) = v$ for some $v \neq v_{low}$ (property DV5). From the fact that $t_s < t_m$, $\texttt{version}_X(t_s) = v_{low}$, $\texttt{version}_X(t_m) = v$, and $v_{low} \neq v$, we can infer that $v_{low} < v$ (property DV3). We thus have to consider two cases with regard to the value of v compared to the value of v_{high}.

If $v \geq v_{high}$, then in state t_m, node X's `version` has already reached or exceeded v_{high}, that is, $\texttt{version}_X(t_m) \geq v_{high}$. Moreover, as previously, since X's `version` never decreases (property DV3), for all $t' \geq t_m$, we have $\texttt{version}_X(t') \geq v_{high}$. Therefore, t_f indeed exists and is equal to t_m.

If, in turn, $v_{high} > v > v_{low}$, then $\texttt{version}_X(t_m) = v$ while our previous assumption still holds: for every state $t \geqslant t_m$, there exists $n \in \texttt{neighborset}_X(t)$, such that $n.\texttt{version}(t) \geqslant v_{high}$ and $n.\texttt{reachable}(t) = \texttt{true}$. Consequently, since $v_{high} > v > v_{low}$ and $v_{high} - v_{low} = i + 1$, we have $v_{high} - v \leqslant i$, and hence can use the inductive assumption (i.e. taking $t_s \leftarrow t_m$ and $v_\Delta \leftarrow v_{high} - v$). What we get in effect is that there exists some state, $t_f \geqslant t_m > t_s$, such that, starting from this state, X's $\texttt{version}$ is forever no less than v_{high}, that is, for all $t' \geqslant t_f$, we have $\texttt{version}_X(t') \geqslant v_{high}$.

These two cases thus conclude the inductive step. The proofs of the inductive base and the inductive step, combined with the fact that the initial state, t_S, the node, X, and the considered computation have been chosen arbitrarily, confirm that the lemma holds for our model. □

Example of a global liveness property. Let us return to Lemma 4, namely to its formulation, which exemplifies that a liveness property is typically conditional: given some assumptions, a desired effect ("something good") will occur. As the lemma describes a local property, its assumptions are simply copied from a property defined in our model (DV5). However, this need not always be the case. In particular, for a global property, one normally needs some additional assumptions on the entire system. Let us thus first demonstrate how such assumptions can be formalized. As a running example, we aim to prove a global property in which the desired effect is—*informally*—the following: "all nodes eventually join (adopt) a DODAG version generated by the root node."

To start with, we need the aforementioned universal assumptions stating that the communication graph, G_{COM}, is finite and connected. In contrast, if the number of nodes were infinite, propagating the generated version would never finish. For a similar reason, if some nodes were disconnected from the rest, they would never get a chance to learn about the generated version.

Pursuing this line of reasoning further, we may observe that learning about a DODAG version depends not only on the existence of links between nodes but also on whether the nodes and links are live, and hence we introduce assumptions A1 and A2. Finally, the root node may generate multiple DODAG versions, some of which will not (and need not) be adopted by all nodes, but we do want to have some guarantees nevertheless. We deal with this problem by formalizing DODAG version generation by the root node as assumption A3.

A1: *Eventually always, all nodes are live.*

A2: *Eventually always, all links are live.*

A3: *Eventually always, the root node does not generate a new value for its* `version`.

The formulation of assumptions A1–A3 borrows from a standard approach to analyzing eventually consistent distributed algorithms:[49,51] assume that the system becomes and remains quiescent (the "eventually always" in A1–A3) and show it will eventually become consistent as well. Demanding system quiescence, assumptions A1–A3 may seem strong at first, but this impression is wrong if they are interpreted correctly. The "eventually always" should be interpreted as "from some moment in time for a sufficiently long period," where "sufficiently long" depends on the actual timings, which are not modeled in LTL. In particular, propagating a DODAG version in RPL can in practice take as little as a few hundred milliseconds, even in large networks. Furthermore, "all nodes/links" in A1 and A2 could be relaxed to a spanning tree, which in addition need not have all its parts simultaneously live. This, however, would require an extended discussion, which we would like to avoid as being immaterial. Consequently, we adopt this standard, quiescence-based formulation of the assumptions and formalize our target global liveness property as Lemma 5. Since assumptions A1–A3 are not universal but rather specific to the particular operational scenario considered in the lemma, we state them explicitly in the lemma body.

Lemma 5. If assumptions A1, A2, and A3 hold, then eventually always, each node has its $\texttt{version}$ equal to the root node's $\texttt{version}$.

Even disregarding its assumptions, the fact that the lemma is a global liveness property should be apparent. It describes the expected behavior of all nodes. Its desired effect is in turn the adoption by the nodes of the last DODAG version generated by the root node.

One of the main challenges in proving such a property is that it is often impossible to conduct the proof on a snapshot of the system (i.e. a single state of a computation involving the system). On the contrary, since the configuration of the system is dynamic, such proofs

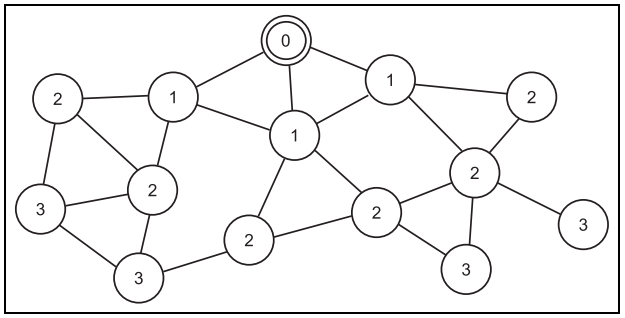

Figure 6. The node's hop counts from the root node in the sample network from Figure 1.

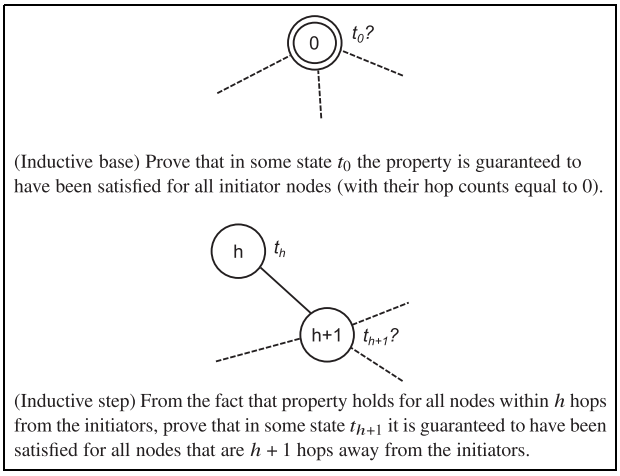

Figure 7. Verifying a liveness property by induction on hop count in the communication graph.

usually analyze what happens in multiple specific consecutive states of the computation. Identifying a sequence of such important states need not be straightforward. In our experience, however, an effective approach is to match the analysis in the proof with the way the desired effect spreads in the system.

In particular, a good initial heuristic is trying to prove such a property by induction on hop count in the communication graph, $G_{COM} = (\mathbb{N}odes, \mathbb{L}inks)$, from the nodes that initiate the desired effect. The communication graph is fixed and assigning each node a hop count computed by the breadth-first search (BFS) algorithm run from a well-defined set of initial nodes yields a partial order suitable for induction (see Figure 6 for an example).

To conduct such a proof, we start by taking an arbitrary computation that satisfies the assumptions of the property: in the case of Lemma 5, assumptions A1–A3. Then, as the inductive base, we need to prove that the property holds at the initiating nodes, for which the hop count is 0: for our lemma, the root node. Typically, this requires finding a state, t_0, of the computation in which the property has been satisfied at the initiator nodes (cf. Figure 7). Finally, as the inductive step, from the fact that the property holds for all nodes within $h \geqslant 0$ hops from the initiators, we have to derive that it also holds for all nodes within $h + 1$ hops. Again, although the details may vary depending on the property, this normally requires demonstrating the existence of a state, t_{h+1}, in which the property is guaranteed to have been satisfied at all nodes that are $h + 1$ hops away from an initiator. We illustrate this technique in the proof of Lemma 5.

Proof. For our model, consider an arbitrary computation that satisfies assumptions A1, A2, and A3.

From assumption A1, there exists a state of the computation, $t_{nodes} \in \{0, 1, 2, \ldots\}$, such that, starting from this state, all nodes are always live. Similarly, from assumption A2, there exists a state of the computation, $t_{links} \in \{0, 1, 2, \ldots\}$, such that, starting from this state, all links are always live. From assumption A3, in turn, there exists a state of the computation, $t_{root} \in \{0, 1, 2, \ldots\}$, such that, starting from this state, the root node never generates a new value for its variable `version`.

Let, thus, $t_0 = max(t_{nodes}, t_{links}, t_{root})$, that is, starting from t_0 all nodes and links are always live and the root node never generates a new value for its variable `version`. We will denote v_{final} as the value of the root node's variable `version` in state t_0, that is, $v_{\text{final}} =$ `version` $_{root}(t_0)$.

We will use induction on the hop count from the root node to prove that, for any $h \geqslant 0$, for all nodes that are up to h hops away from the root, there exists a state, $t_h \geqslant t_0$, starting from which the nodes forever have their variables `version` equal to v_{final}.

To this end, as the inductive base, consider $h = 0$. The only node at this distance from the root node is the root node itself. From the definition of state t_0, in turn, we know that in this state the root node's `version` equals to v_{final}, that is, `version` $_{root}(t_0) = v_{\text{final}}$. Moreover, in no state $t \geqslant t_0$, does the root node generate any new value for its `version`. Therefore, the only way for the root node to change its `version` from v_{final} to some other value, v', would be by adopting that value (property DV4). We will prove by contradiction that this is not possible.

Accordingly, assume that a transition of the system from some state $t' \geqslant t_0$ to state $t' + 1$ corresponds to the root node adopting some $v' \neq v_{\text{final}}$ as its `version`, that is, `version` $_{root}(t') = v_{\text{final}}$ while `version` $_{root}(t' + 1) = v'$. Since $v' \neq v_{\text{final}}$, we must have $v' > v_{\text{final}}$ (property DV3). The transition also implies that, in state t', the root node's `neighborset` contained an entry, `n`, which had its field `version` equal to v', that is, `n` $\in$ `neighborset` $_{root}(t')$ and `n.version` $(t') = v'$ (property DV4). Combining these two facts we thus get that in state t', in which the root node was live, in the root node's `neighborset` there was an entry, `n`, which had its field `version` greater than the value of the root node's variable `version`, that is, `n`$\in$`neighborset` $_{root}(t')$ and `n.version` $(t') = v' > v_{\text{final}} =$ `version` $_{root}(t')$. This, however, contradicts Lemma 3, which asserts that `n.version` $(t') \leqslant$ `version` $_{root}(t')$. In other words, the considered transition is impossible, and hence indeed, in all states $t \geqslant t_0$, we must have `version` $_{root}(t) = v_{\text{final}}$, which establishes the inductive base.

For the inductive step, assume that for all nodes that are up to $h \geqslant 0$ hops away from the root node, there exists a state, $t_h \geqslant t_0$, such that, starting from this state,

the nodes' `version` variables forever equal v_{final}. We will show that for all nodes that are up to $h + 1$ hops away from the root node, there exists a similar state, $t_{h+1} \geqslant t_0$, starting from which, the nodes' `version` variables are always equal to v_{final}.

To this end, consider an arbitrary node, X, that is $h + 1$ hops away from the root node. We will first show that for this node, there exists a state, $t_{h+1,X} \geqslant t_h$, starting from which the node's variable `version` is equal to v_{final}, that is, for all $t \geqslant t_{h+1,X}$, `version` $_X(t) = v_{\text{final}}$.

To begin with, from the definition of t_h and the fact that X is $h + 1$ hops away from the root node, we know that in all states $t \geqslant t_h$, node X is adjacent to some node, Y, that is h hops away from the root and for which `version` $_Y(t) = v_{\text{final}}$. This has two important implications.

First, node X's `neighborset` eventually always contains an entry corresponding to node Y and having its `reachable` flag set to `true` (property RA4), that is, there exists some state, $t_{X,reach} \geqslant t_h$, such that in all states $t \geqslant t_{X,reach}$, there exists $n \in$ `neighborset` $_X(t)$ such that `n. id` $(t) = Y$ and `n. reachable` $(t) =$ `true`.

Second, from t_h, node X is guaranteed to always eventually receive a DIO message from node Y (property CT2). Field `version` of each such message, `d`, is equal to the value of `version` $_Y$ in the state in which `d` was transmitted by Y (property CT1). Messages that had been transmitted by Y before its `version` became v_{final} may still be in transit after t_h, and hence may be received by X in some states following t_h. However, eventually always, the reception of such messages by X does not occur (property CT3), that is, there exists some state, $t_{X,dio}$, such that any DIO message, `d`, received by X from Y in any state $t \geqslant t_{X,dio}$ satisfy the condition: `d. version` $(t) = v_{\text{final}}$.

Since, as mentioned previously, from t_h, X always eventually receives a DIO message from Y, let $t_{X,field}$ be the first state numbered at least $max(t_{X,dio}, t_{X,reach})$ in which such a message, d, is received. From the definition of $t_{X,field}$, we are guaranteed in this and all subsequent states what follows. First, there exists in X's *neighborset* an entry, `n`, corresponding to node Y and having its `reachable` flag set to `true`. Second, field `version` of this entry is equal to v_{final}, as this is the value of the same field in `d` and any following DIO message received by X from Y (property RA2). Put differently, in any state $t \geqslant t_{X,field}$, there exists $n \in$ `neighborset` $_X(t)$ such that `n. reachable` $(t) =$ `true` and `n. version` $(t) = v_{\text{final}}$. Let us analyze two cases depending on the relation between `version` $_X(t_{X,field})$ and v_{final}.

If `version` $_X(t_{X,field}) \geqslant v_{\text{final}}$, then in all states $t \geqslant t_{X,field}$, `version` $_X(t) \geqslant v_{\text{final}}$ (property DV3). Let us thus denote $t_{X,final} = t_{X,field}$.

If, in turn, `version` $_X(t_{X,field}) < v_{\text{final}}$, then, from the definition of state $t_{X,field}$, the premises of Lemma 4 are satisfied in this state, with $v_{low} =$ `version` $_X(t_{X,field})$ and $v_{high} = v_{\text{final}}$. Consequently, the lemma entails that there exists some state, $t_{X,lemma}$, such that in all states $t \geqslant t_{X,lemma}$, `version` $_X(t) \geqslant v_{\text{final}}$. Let us thus denote $t_{X,final} = t_{X,lemma}$.

To recap, in both cases, there exists a state, $t_{X,final}$, such that in all states $t \geqslant t_{X,final}$, `version` $_X(t) \geqslant v_{\text{final}}$. Moreover, from the inductive assumption, in any state $t \geqslant t_{X,final} \geqslant t_h$, we have `version` $_{root}(t) = v_{\text{final}}$. At the same time, from Lemma 3, we know that in particular for every state $t \geqslant t_{X,final}$, `version`$_X(t) \leqslant$ `version` $_{root}(t)$. Combining these three facts, we conclude that in all states $t \geqslant t_{X,final}$, we must have `version` $_X(t) = v_{\text{final}}$. In other words, we can take $t_{h+1,X} = t_{X,final}$ as the sought state, starting from which node X's `version` is forever equal to v_{final}.

Since node X was chosen arbitrarily, the same holds for any node that is $h + 1$ hops away from the root. Therefore, we can take $t_{h+1} = max_{\{X \in \text{Nodes} \mid X \text{ is } h+1 \text{ hops away from the root node}\}}(t_{h+1,X})$, as the state starting from which all nodes that are $h + 1$ hops away from the root node forever have their variables `version` equal v_{final}, which completes the inductive step.

Combining the inductive base and the inductive step with the fact that the communication graph formed by nodes and links is finite and connected, and hence the maximal distance in hops of any node from the root node is bounded, we can conclude that eventually always each node in the considered computation has its `version` equal to the root node's `version`. Finally, since the computation was chosen arbitrarily among all computations satisfying assumptions A1, A2, and A3, Lemma 5 simply holds. □

Devising custom non-trivial inductive orders

While induction on hop count may be a good starting point when trying to prove a global liveness property, it need not work in all cases. As an illustration, let us consider Hypothesis 1, whose simplified version was originally subject of our previous work.[7]

Hypothesis 1: *If A1, A2, and A3 hold, then eventually always, all nodes have their* `ranks` *finite.*

The desired effect in the hypothesis concerns RPL's fundamental functionality: building a DODAG that allows all nodes to perform upward routing to the root node. To be precise, for a DODAG to be formed, every non-root node must have its `prefpar` non-`null`. However, to simplify our reasoning by uniformly treating the root node and non-root nodes, we consider `rank` s instead of `prefpar` s in the hypothesis. This is

because a non-root node has its `prefpar` non-`null` iff its `rank` is finite (properties OF4 and OF1), whereas the root node's `prefpar` is always `null` while its `rank` is always finite (properties OF3 and OF1). As to the assumptions, the motivation behind A1 and A2 is the same as for Lemma 5. Assumption A3 is in turn dictated by the fact that a DODAG version change resets all non-root nodes' `rank` s and `prefpar` s (property OF4), and hence, if a given version does not exist sufficiently long, all nodes will not be able to join it and select their preferred parents and ranks appropriately.

We leave to the interested readers' discretion an attempt to prove Hypothesis 1 by induction on hop count in the communication graph. In summary, under an additional assumption, which we will return to shortly, it is relatively straightforward to prove the inductive base and show that if all nodes within $i \geqslant 0$ hops from the root eventually always have their `rank` s finite, then a node within $i + 1$ hops also eventually has its `rank` finite.[7] Although the latter may seem as the inductive step, there is a subtle issue: the conclusion states "eventually has its `rank` finite" rather than "eventually *always* has its `rank` finite." The lack of "always" implies that the node within $i + 1$ hops may not forever keep its `rank` finite, which would be necessary for proving the inductive step. In contrast, proving the actual inductive step need not be possible. The reason is that because of continuous changes in the routing `metrics` of the node's neighbors, in some states of the computation, the node may not be able to select as its `prefpar` a neighbor that is up to i hops away from the root; it may be able to select a neighbor that is, for instance, $i + 2$ hops away, though.

This observation hints at the main problem with hop count as the inductive order for a proof of Hypothesis 1: parent selection by the objective function depends on *dynamic* routing metric values and hence may not reflect the nodes' *static* hops from the root. In particular, if at some point a neighbor with a larger hop count offers a node a better potential rank (r in property OF5) than a neighbor with a smaller hop count, then the node can select the former as its `prefpar`. As an example, consider the node with rank 4.3 and its neighbors with ranks 2.1 and 3.3 in Figure 1. Therefore, an intuitive induction order seems to be the one induced by the preferred parent relation.

There are two problems with this relation, though. First, the subgraph induced by the nodes' links to preferred parents need not be acyclic, which is necessary for a graph describing an order. Second, the node's preferred parents can change in subsequent states of a computation, whereas inductive reasoning requires an order to be fixed. As a result, we cannot utilize the preferred parent relation as an order for induction.

To propose an alternative order, let us introduce a concept of what we dubbed *best ever parents* (*BEP*s). A node's BEP is defined for a given DODAG version and is the node's current or former preferred parent, whose selection gave the node the smallest rank ever in this particular DODAG version for the first time. In our notation, this is the node's `prefpar` whose selection led to the last drop in the node's `minrank` after the node's `version` had been modified for the last time or after the node had started for the first time, if its `version` had never been modified (see Figure 8).

Given this concept, in any state, $t \in \{0, 1, 2, \ldots\}$, of a computation involving our model, for any given DODAG version, v, either one that was generated before t or may yet be generated after t, we define a directed *BEP graph*, $G_{BEP(v)}(t) = (V_{BEP(v)}(t), E_{BEP(v)}(t))$, which is a subgraph of the communication graph G_{COM}. More specifically, its set of vertices is simply the set of all nodes, that is, $V_{BEP(v)}(t) = \mathbb{N}\text{odes}$. Its set of edges, in turn, contains links from non-root nodes to their BEPs in DODAG version v in state t, that is, $E_{BEP(v)}(t) = \{\langle X, Y\rangle \in \mathbb{L}\text{inks} \mid$ in state t, Y is X's BEP in DODAG version $v\}$. The following lemmas describe the properties of the BEP graph.

Lemma 6. Always, the best-ever-parents (BEP) graph for any DODAG `version`, v, $G_{BEP(v)} = (V_{BEP(v)}, E_{BEP(v)})$ is finite and composed out of two disjoint subgraphs: a directed tree $G_{BEP-T(v)} = (V_{BEP-T(v)}, E_{BEP-T(v)})$, having the root node as the only sink and comprising non-root nodes that have had a BEP within DODAG `version` v, and an unconnected graph, $G_{BEP-U(v)} = (V_{BEP-U(v)}, E_{BEP-U(v)})$, containing the remaining nodes. In other

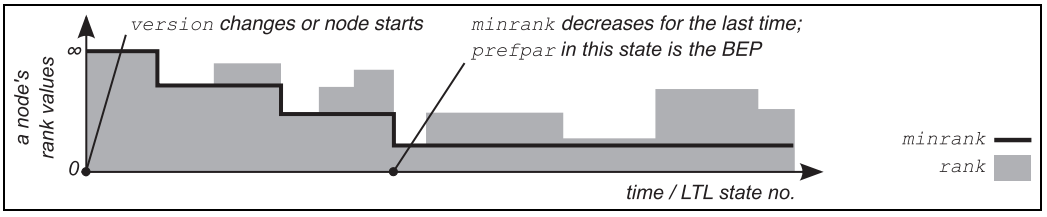

Figure 8. An illustration of the best ever parent (BEP) idea.

words, in any state $t \in \{0, 1, 2, ...\}$ of any computation involving our model, $E_{BEP-T(v)}(t) = E_{BEP(v)}(t)$, and $V_{BEP-T(v)}(t) = \{X \in \mathbb{N}\text{odes} \mid X \text{ is the root or there exists a link } \langle Y, X\rangle \in E_{BEP(v)}(t) \textit{ for some } Y \in \mathbb{N}\text{odes}\}$, whereas $V_{BEP-U(v)}(t) = V_{BEP(v)}(t) \setminus V_{BEP-T(v)}(t) = \mathbb{N}\text{odes} \setminus V_{BEP-T(v)}(t)$ and $E_{BEP-U(v)}(t) = E_{BEP(v)}(t) \setminus E_{BEP-T(v)}(t) = \emptyset$.

Lemma 7. The best-ever-parents (BEP) graph for any DODAG version, v, $G_{BEP(v)} = (V_{BEP(v)}, E_{BEP(v)})$ eventually always does not change. In other words, in any computation involving our model, there exists some state $t_{BEP(v)} \in \{0, 1, 2, \ldots\}$, such that for any $t \geqslant t_{BEP(v)}$, $G_{BEP(v)}(t) = G_{BEP(v)}(t_{BEP(v)})$.

The proofs of these lemmas (merged into a single lemma) can be found in our earlier paper.[7] The interested readers should likely be able to recreate them. To give some hints, proving Lemma 6 requires showing two properties, which in both cases can be done by contradiction: first, that the BEP graph never contains any cycle and second, that any path in the graph ends at the DODAG root. Proving Lemma 7, in turn, boils down to observing that a node's *minrank* is an integer with a lower-bound formalized in Lemma 1 and hence cannot decrease indefinitely.

From our perspective, what is more important than the details of the proofs themselves is the implications of the lemmas. Lemma 6 essentially states that, in any state of any computation, the BEP graph for any DODAG version is acyclic and hence can represent a partial order of nodes. Lemma 7 implies in turn that from some state of any computation this order is fixed. Combining the lemmas, we can thus conclude that the order induced by the BEP graph is suitable for proofs by mathematical induction. An example of a possible BEP graph, and the induced order, for the sample DODAG from Figure 1 is presented in Figure 9.

Such a proof proceeds roughly in the same manner as in the previous techniques but involves some extra steps. Like usual, we start by taking an arbitrary computation that satisfies the assumptions of the property that is being proved. Then, we consider a specific DODAG version, v, and state, $t_{BEP(v)}$, in which the BEP graph for this DODAG version becomes fixed (cf. Lemma 7) as well as succeeding states. To establish the inductive base, we need to show that in some or all such states (depending on the property), the property holds for the sink of the BEP graph, that is, the root node. As the inductive step, in turn, we take an arbitrary non-root node and from the fact that the property holds for the node's BEP in some or all of the considered states (again, depending on the property), we have to derive that it also holds for the node itself. At this point, mathematical induction allows us to claim that the property holds for the root node and all non-root nodes that have a BEP in the considered DODAG version, that is, it holds for the subgraph $G_{BEP-T(v)} = (V_{BEP-T(v)}, E_{BEP-T(v)})$ of the BEP graph (cf. Lemma 6). In turn, to ascertain that the property holds for all nodes—not only for those in the subgraph—we can show, for instance, that there is no node in the considered DODAG version that does not have a BEP, that is, $V_{BEP-U(v)} = \emptyset$ (cf. Lemma 6).

As we demonstrate next, following this approach allows for proving what we tried to express in Hypothesis 1: that RPL is able to build and maintain a DODAG. From a broader perspective, in turn, our discussion hitherto exemplifies how one can devise a dedicated inductive order to match the way the desired effect of a global liveness property spreads in the system. Finally, the presented theoretical concept of BEPs also has practical applications, such as monitoring DODAG health for generating DODAG versions, especially for custom routing metrics.[7]

Working with hypotheses and assumptions

Given a technique advertised as suitable for proving that nodes in RPL are capable of choosing and keeping finite ranks, we will proceed slightly differently than previously. Rather than just giving the target lemma, we will present the entire process of formulating and proving it. We will propose an initial version of the

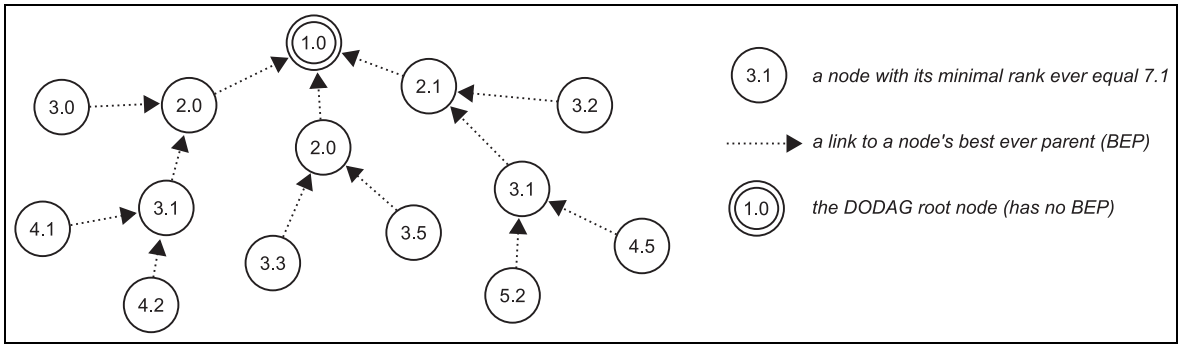

Figure 9. An illustration of the best ever parents in the sample DODAG from Figure 1.

lemma by adding two assumptions and adapting Hypothesis 1 to the devised proof technique. While proving the lemma, we will be reformulating it and introducing further assumptions. This will illustrate what we mentioned previously: that a formal verification process is normally iterative. All in all, this section is meant to complete the suite of skills that we believe are necessary to begin modeling and verifying routing protocols on one's own.

Formulating an initial hypothesis. We start with the missing assumption mentioned in the previous section. More specifically, let us observe that the only way for a non-root node to get its `rank` finite is `prefpar` and `rank` reselection because the other events affecting the node's `rank` (property OF1) make it infinite (property OF4). Yet, the rules of the reselection process (properties OF4 and OF5), are insufficient to guarantee liveness, that is, guarantee that the node would adopt a finite `rank` if specific conditions were met. Indeed, there exist LTL computations that satisfy all properties of our model and assumptions A1–A3 but in which the potential ranks, `r`, calculated by non-root nodes during `prefpar` and `rank` reselection are always infinite (cf. property OF5). In effect, in such computations, non-root nodes never have their `rank`s finite. In other words, without additional liveness conditions for computing potential ranks, Hypothesis 1 is false. Since RPL's specification does not explicitly define such conditions, entirely delegating the computation of nodes' potential ranks to objective functions, we did not include them among the OF properties in our model. Consequently, we address this issue now by introducing assumption A4.

A4: *Always, if a non-root node reselects its `prefpar` and `rank`, then for each entry, `n`, in its neighborset that satisfies conditions (a), (b), and (c) of property OF5, the potential `rank`, `r`, computed by the node satisfies conditions (d) and (e) of property OF5.*

What A4 states is that for each neighbor set entry representing a node's neighbor that belongs to the same DODAG version as the node, has itself advertised a finite rank recently, and is considered by the node as reachable, the node is guaranteed to obtain a finite rank, irrespective of the objective function and values of the routing metrics employed. Demanding in this case that a potential rank be finite, albeit possibly very high, is a sensible assumption, further backed by what RPL's aforementioned implementations ensure. Although we could adopt a weaker one, this would require a deeper discussion, which is not strictly relevant and is thus avoided here for brevity.

The second issue is that the current formulations of the OF properties, and our model in general, completely abstract out routing metrics. However, changes to their values do affect calculated potential ranks and hence the nodes' ranks. Therefore, to continue treating routing metrics as "black boxes" while being able to reason about their changes, we introduce assumption A5.

A5: *Always, if a non-root node reselects its `prefpar` and `rank`, the potential `rank`, `r`, computed for each entry, `n`, in the node's neighborset satisfies the following condition: `r = n. rank` + $\rho(n)$, where $\rho(n)$ does not depend on `n. rank`.*

Assumption A5 stems from RPL's recommendation regarding the function for computing potential ranks: the function should be additive. At the same time, through function ρ, the assumption leaves a lot of freedom in how precisely routing metric values contribute to potential ranks computed by an objective function during reselection. In effect, it captures all practical configurations in RPL's both aforementioned open-source implementations. Therefore, we believe it is realistic. What is more, it allows us to treat routing metrics are "black boxes," in particular, to describe their changes for a given entry, `n`, in a node's `neighborset` in terms of changes to the values of $\rho(n)$.

Given A4 and A5 we are ready to revise the false Hypothesis 1, reformulating it into Hypothesis 2. Apart from including the two assumptions, the new hypothesis explicitly foresees that more may yet be necessary. In addition, it considers only the members of the final BEP tree, $G_{BEP-T(v_{\text{final}})}$, for which the inductive technique that we devised in the previous section can be applied; the isolated nodes, forming the other subgraph of the final BEP graph, $G_{BEP-U(v_{\text{final}})}$, are in turn disregarded.

Hypothesis 2: *If assumptions A1–A5 and possibly some yet unforeseen ones hold, then eventually always, each member of BEP tree $G_{BEP-T(v_{\text{final}})}$ has its `rank` finite, where v_{final} is the last DODAG `version` at the root node in accordance with assumption A3.*

Refining the hypothesis. If we introduced yet another assumption, A6, proving Hypothesis 2 would be relatively easy, which we leave as an exercise to the interested readers.

A6: *For each entry, `n`, in a node's `neighborset`, eventually always $\rho(n)$ is equal to its smallest value ever in a given DODAG `version`, that is, its smallest value since the node's `version` last changed or*

since the node started for the first time, if its `version` *has never changed.*

Assumption A6 makes it possible for the routing metrics for a neighbor—or, given assumption A5, the corresponding values of $\rho(n)$—to grow arbitrarily. Nevertheless, it also requires them to ultimately converge to their minimal values in a given DODAG version and remain such forever. In combination with assumptions A1–A5, this ensures that eventually always a node's BEP is a suitable candidate for the node's preferred parent according to property OF5. In particular, the potential rank, `r`, it offers the node is within the node's rank limit defined by OF5 5.

However, assumption A6 is too strong to be satisfiable in practice: routing metrics rarely remain fixed at their minimal values; on the contrary, usually they change dynamically. Consequently, for our work to have any real-world relevance, this assumption must be weakened, which implies allowing routing metrics to grow. However, considering the previous argument, if this growth were unbounded, then it would be possible for a node to always lack a neighbor entry satisfying OF5(f). We thus have to somehow limit the growth of the routing metric values.

To this end, for each neighbor entry, `n`, we assume that $\rho(n)$ is always within some constant, ρ_Δ, from its minimal value in a given DODAG version. Despite not being stated in RPL's specification, this requires `MaxVersionRankIncrease` to depend on this constant and, what may be even more unexpected, also on the global network topology, which is again not mentioned in the specification. What is even worse, the specification may suggest the opposite, implying that `MaxVersionRankIncrease` is for local DODAG repairs.

We encourage the interested readers to come up with these dependencies of `MaxVersionRankIncrease` on their own. As a hint, try to find a scenario in which `MaxVersionRankIncrease` must not be smaller than $\rho_\Delta \times (N-1)$, where N is the node population size.[7] A more precise formulation of the dependencies is in turn expressed by assumption A6, which is a weakened version of assumption A6, that is, A6 replaces A6. As a side note, this exemplifies how formal methods can help designing routing protocols and creating their specifications.

A6′: *For each entry,* `n`*, in a node's* `neighborset`*, eventually always* $\rho(n)$ *does not exceed its smallest ever value in a given DODAG version,* `v`*, by more than* `MaxVersionRankIncrease` $/ D$*, where* D *is the final depth of BEP tree* $G_{BEP-T(v)}$.

Having addressed the liveness of *prefpar* and *rank* reselection as well as the dynamics of routing metric values, it seems we may be ready to formulate and attempt to prove a variant of our target hypothesis, Hypothesis 2, which has stronger conclusions than Hypothesis 2.

Hypothesis 2′: *If assumptions A1–A5, A6 and possibly some yet unforeseen ones hold, then eventually always, each member of the final BEP tree,* $G_{BEP-T(v_{\text{final}})}$*, has its* `rank` *finite, not exceeding its* `minrank` *in DODAG* `version`v_{final} *by more than* $d \times$ `MaxVersionRankIncrease` $/ D$*, where* d *is the node's depth in the BEP tree,* D *is the final depth of the BEP tree itself, as defined in assumption A6, and* v_{final} *is the last DODAG* `version` *at the root node in accordance with assumption A3.*

Proof. For our model, consider an arbitrary computation that satisfies assumptions A1–A5 and A6. From assumptions A1–A3 and Lemma 5, there exists a state, $t_{version}$, starting from which all nodes and links are live, no new DODAG versions are generated, and all nodes have their variables *version* equal to the root's variable. Note thus that $v_{\text{final}} = \texttt{version}_{root}(t_{version})$ is the final DODAG version. From Lemma 7, there exists a state, $t_{BEP(v_{\text{final}})}$, such that, starting from this state, the BEP graph for the final DODAG version, $G_{BEP(v_{\text{final}})}$, does not change, that is, for any $t \geqslant t_{BEP(v_{\text{final}})}$, we have $G_{BEP(v_{\text{final}})}(t) = G_{BEP(v_{\text{final}})}(t_{BEP(v_{\text{final}})})$. Let us denote $t_0 = max(t_{BEP(v_{\text{final}})}, t_{version})$. Since from Lemma 6 for any t, $G_{BEP(v_{\text{final}})}(t)$ defines a partial order on nodes, we will perform induction over the final BEP tree for DODAG version v_{final}, that is, $G_{BEP-T(v_{\text{final}})}(t_0)$, which also means that we will analyze the system states starting from t_0.

As the inductive base consider the root node, whose depth, d, in the BEP tree, $G_{BEP-T(v_{\text{final}})}(t_0)$, is 0. For any $t \geqslant 0$, we have $\texttt{rank}_{root}(t) = \texttt{MinHopRankIncrease}$ (properties OF1 and OF3). Therefore, for any $t \geqslant 0$, we also have $\texttt{minrank}_{root}(t) = \texttt{MinHopRankIncrease}$ (property OF2). Combining these two facts, for all $t \geqslant t_0$, we get $rank_{root}(t) \leqslant \texttt{minrank}_{root}(t) + 0 \times \texttt{MaxVersionRankIncrease} / D < \infty$, which establishes the inductive base.

To show the inductive step, in turn, consider an arbitrary non-root node, X, with depth $d = i + 1$ in the final BEP tree, $G_{BEP-T(v_{\text{final}})}(t_0)$, and its BEP in the tree, node Y, with depth i, where $0 \leqslant i < D$. Our inductive assumption is that there exists some state, $t_Y \geqslant t_0$, such that in all states $t \geqslant t_Y$, $\texttt{rank}_Y(t) \leqslant \texttt{minrank}_Y(t) + i \times \texttt{MaxVersionRankIncrease} / D < \infty$. We have to show that there exists an analogous state, $t_X \geqslant t_0$, such that in all states $t' \geqslant t_X$, $\texttt{rank}_X(t') \leqslant \texttt{minrank}_X(t') + (i+1) \times \texttt{MaxVersionRankIncrease} / D < \infty$.

We start by showing that eventually always, whenever X calculates—in compliance with OF5 —the potential rank for the entry corresponding to Y in its `neighborset`, then this potential rank, which we denote as $r_{X,Y}$, does not exceed X's *minrank* by more than $(i+1) \times$ `MaxVersionRankIncrease` $/ D$. In other words, we will show that there exists some state $t_{X,Y} \geq t_0$ such that for any state $t \geq t_{X,Y}$, we have $r_{X,Y}(t) \leq minrank_X(t) + (i+1) \times$ `MaxVersionRankIncrease` $/ D$. To this end, let us make the following observations.

(i) From t_0 on, nodes X and Y are forever adjacent. Consequently, there exists a state, $t_{X,Y,reach} \geq t_0$, starting from which the entry corresponding to Y in X's `neighborset`, which we denote as $n_{X,Y}$, always exists and has its `reachable` flag set to `true` (property RA4), that is, for all states $t \geq t_{X,Y,reach}$, we have $n_{X,Y} \in neighborset_X(t)$ and $n_{X,Y}$. `reachable(t) = true`.

(ii) From t_0 on, $version_X = version_Y = v_{\text{final}}$. Therefore, there exists a state, $t_{X,Y,ver} \geq t_0$, starting from which field `version` of entry $n_{X,Y}$ is also v_{final} (properties CT1, CT2, CT3, and RA2), that is, in all states $t \geq t_{X,Y,ver}$, we have $n_{X,Y}$. `version` (t) = $version_X(t) = v_{\text{final}}$.

(iii) From t_Y on, $rank_Y$ is finite and does not exceed $minrank_Y$ by more than $i \times$ `MaxVersionRankIncrease` $/ D$ (inductive assumption). As a result, there exists a state, $t_{X,Y,rank} \geq t_Y$, starting from which also field `rank` of entry $n_{X,Y}$ satisfies this condition (properties CT1, CT2, CT3, OF2, and RA2), that is, in all states $t \geq t_{X,Y,rank}$, we have $n_{X,Y}$. `rank`$(t) \leq minrank_Y(t) + i \times$ `MaxVersion RankIncrease` $/ D < \infty$.

(iv) (iv) There exists a state, $t_{X,Y,metr} \geq t_0$, from which on $\rho(n_{X,Y})$ does not exceed its smallest value in DODAG version v_{final}, which we denote as $\rho_{min(v_{\text{final}})}(n_{X,Y})$ by more than `MaxVersionRankIncrease` $/ D$ (assumption A6), that is, in all states $t \geq t_{X,Y,metr}$, we have $\rho(n_{X,Y})(t) \leq \rho_{min(v_{\text{final}})}(n_{X,Y}) +$ `MaxVersionRankIncrease` $/ D$.

(v) (v) From the fact that node Y is node X's BEP in DODAG version v_{final} and that, in any DODAG version, a node's `minrank` can only decrease (property OF2), from t_0 on, $minrank_X$ is finite and exceeds $minrank_Y$ by at least $\rho_{min(v_{\text{final}})}(n_{X,Y})$, that is, for any state $t \geq t_0$, we have $minrank_Y(t) + \rho_{min(v_{\text{final}})}(n_{X,Y}) \leq minrank_X(t) < \infty$.

Consider state $t_{X,Y,pot} = max(t_0, t_{X,Y,reach}, t_{X,Y,ver}, t_{X,Y,rank}, t_{X,Y,metr})$. In any state $t \geq t_{X,Y,pot}$, the entry corresponding to node Y, $n_{X,Y}$, is guaranteed to exist in node X's `neighborset` (observation 1), and satisfy conditions 5 (obs. 2), 5 (obs. 3), 5 (obs. 1), and both 5 and 5 (assumption A4) of property OF5. Moreover, the potential rank for this entry, $r_{X,Y}$, has to satisfy what follows (where $M =$ `MaxVersionRankIncrease`):

$$
\begin{aligned}
r_{X,Y}(t) &\overset{A5}{=} n_{X,Y}.\,rank\,(t) + \rho(n_{X,Y})(t) \\
&\overset{obs.\,3}{\leq} minrank_Y(t) + i \times M / D + \rho(n_{X,Y})(t) \\
&\overset{obs.\,4}{\leq} minrank_Y(t) + \rho_{min(v_{\text{final}})}(n_{X,Y}) + (i+1) \times M / D \\
&\overset{obs.\,5}{\leq} minrank_X(t) + (i+1) \times M / D < \infty.
\end{aligned}
$$

Since $i < D$, and hence $(i+1) \times$ `MaxVersion RankIncrease` $/ D \leq$ `MaxVersionRankIncrease`, this also implies that $r_{X,Y}$ satisfies OF5 5. All in all, we have just proved that from $t_{X,Y,pot}$ on, whenever Y, which is X's BEP, is also X's `prefpar`, then X's `rank`, assigned from $r_{X,Y}$, satisfies the conclusions of the lemma.

As the next step, we will prove that node X has a chance to choose node Y as its `prefpar`, and hence to adopt $r_{X,Y}$ as its `rank`, that is, it performs `prefpar` and `rank` reselection at least once after $r_{X,Y}$ starts satisfying the conclusions of the lemma. To this end, let us observe that any of the events that affect X's `neighborset` is followed by such a reselection (RA1). In particular, the event affecting $n_{X,Y}$ as the one that makes $r_{X,Y}$ start satisfying the conclusions of the lemma is followed by a reselection. Since, from that event, $r_{X,Y}$ continues satisfying the conclusions of the lemma forever, during the reselection, X can indeed choose Y as its `prefpar` and adopt $r_{X,Y}$ as its `rank`.

To recap, we have proved that eventually whenever Y, which is X's BEP, is also X's `prefpar`, then X's `rank` satisfies the conclusions of the lemma, and that X has at least one chance to choose Y as its `prefpar`, and hence adopt such a `rank`. Therefore, if node X takes this chance, choosing Y as its `prefpar`, and keeps this choice in all succeeding states, then indeed its `rank` eventually always satisfies the conclusions of the lemma. However, no property or assumption *forces* X to do so. In particular, node X may always eventually (i.e. from time to time) select as its `prefpar` another node that makes its `rank` exceed its *minrank* $+ (i+1) \times$ `MaxVersionRankIncrease` $/ D$. To prevent this, we thus have to introduce another rule of `prefpar` and `rank` reselection, such as assumption A7.

A7: *Always, if a node selects a finite* `rank` *and non-null* `prefpar`*, then its* `neighborset` *does not contain an entry,* `n`*, that satisfies all conditions (a)–*

(f) of property OF5 and that offers the node a potential `rank`*,* `r`*, smaller than the selected* `rank`*.*

A7 simply states that a node always selects as its preferred parent the neighbor that offers it the lowest potential rank. This can be the node's BEP or another neighbor, if during a particular reselection that neighbor offers the node a smaller rank than the BEP. For node X, if adopting the potential rank offered by its BEP, Y, already guarantees that X's *rank* does not exceed X's *minrank* by more than $(i + 1) \times$ `MaxVersionRankIncrease` $/ D$, then so does adopting an even smaller rank. In other words, with A7, taking $t_X = t_{X,Y,pot}$ completes the proof of our inductive step, because for all $t' \geqslant t_X$, we have $rank_X(t') \leqslant minrank_X(t') + (i + 1) \times$ `MaxVersionRankIncrease` $/ D < \infty$.

However, instead of concluding the entire proof, let us observe that despite not forcing the objective function to select a node's BEP as the preferred parent, A7 may still be too constraining. This is because it always requires the objective function to select a preferred parent that offers the node the lowest rank. Since always aiming at the best preferred parent may result in frequent parent changes, there exist objective functions, such as MRHOF,[56] that explicitly rely on more freedom in the choice of a node's preferred parent. This freedom is realized by employing the aforementioned hysteresis, which prohibits the node from switching its preferred parent unless this is sufficiently beneficial, that is, unless the potential rank offered by the new parent is lower than the node's current rank by more than a threshold, denoted *ParentSwitchThr*. Consequently, to capture such freedom in our model, we have to relax A7, replacing it with assumption A7.

A7′:*Always, if a node selects a finite* `rank` *and non-*`null` `prefpar`*, then its neighborset does not contain an entry,* `n`*, that satisfies all conditions (a)–(f) of property OF5 and that offers the node a potential rank,* `r`*, smaller than the selected rank by more than ParentSwitchThr.*

While A7—and hysteresis in general—gives much more freedom when selecting preferred parents, it may lead to their suboptimal choices if *ParentSwitchThr* $>$ 0. To explain, under the previous assumptions, if selecting a node's BEP as `prefpar` guaranteed that the node's *rank* would not exceed `minrank` by a margin larger than $(i + 1) \times$ `MaxVersionRankIncrease` $/ D$, then selecting a neighbor offering a potential rank greater by up to `ParentSwitchThr`, would make the margin grow by `ParentSwitchThr`, which would violate the conclusions of the lemma. What is more, such a growth would accumulate with nodes' depth in the BEP tree, as at each hop, the `rank` resulting from the choice of a corresponding node's `prefpar` may be suboptimal up to `ParentSwitchThr`. At the same time, this latter observation gives a hint on how our assumptions could be patched: the tolerated routing metric values oscillations have to take into account hysteresis in parent selection. More precisely, we can replace assumption A6 with a stronger one, A7, which also indicates that `ParentSwitchThr` must not exceed `MaxVersionRankIncrease` $/ D$.

A6″:*For each entry,* `n`*, in a node's* `neighborset`*, eventually always* $\rho(n)$ *does not exceed its smallest ever value in a given DODAG* `version`*,* `v`*, by more than* `MaxVersionRankIncrease` $/ D -$ `ParentSwitchThr`*, where D is the final depth of BEP tree* $G_{BEP-T(v)}$*.*

Apart from taking A7 instead of A6, our previous reasoning remains valid otherwise, verifying which we leave to the interested readers. In other words, the inductive step holds with $t_X = t_{X,Y,pot}$.

Combining the inductive base and the inductive step with the fact that the final BEP tree, $G_{BEP-T(v_{\text{final}})}$, is finite and connected, and hence its maximal depth, D, is bounded, we can conclude that in the considered computation, eventually always, each node belonging to the tree has its `rank` finite, not exceeding its `minrank` by more than $d \times$ `MaxVersionRankIncrease` $/ D$, Finally, since the computation was chosen arbitrarily among all computations satisfying the given assumptions, Hypothesis 2′ is simply true. □

To emphasize this fact, we present the final hypothesis that we have just proved as Lemma 8.

Lemma 8. If assumptions A1, A2, A3, A4, A5, A6″, and A7 hold, then eventually always, each member of the final BEP tree, $G_{BEP-T(v_{\text{final}})}$, has its `rank` finite, not exceeding its `minrank` by more than $d \times$ `MaxVersionRankIncrease` $/ D$, where d is the node's depth in the BEP tree, D is the final depth of the BEP tree itself, as defined in assumption A7, and v_{final} is the last DODAG `version` at the root node in accordance with assumption A3.

Closing remarks. The previous reasoning showed how by iterating on hypotheses and their assumptions we can prove a key property for RPL. In particular, given Lemma 8, we can further show that the final BEP tree eventually always covers all nodes, that is, eventually always $V_{BEP-T(v_{\text{final}})} = V_{BEP(v_{\text{final}})}$, as formalized in Lemma 9. A relatively straightforward proof of the lemma can be done by contradiction, which we again leave to the interested readers.

Lemma 9. If assumptions A1–A5, A6″, and A7′ hold, then eventually always, the final BEP graph contains no isolated nodes, that is, $V_{BEP(v_{\text{final}})} = V_{BEP-T(v_{\text{final}})}$ (or equivalently, $V_{BEP-U(v_{\text{final}})} = \emptyset$).

Finally, by combining Lemmas 6, 7, 8, and 9, we can also conclude what follows.

Lemma 10. If assumptions A1–A5, A6″, and A7′ hold, then eventually always, all nodes have their `rank`s finite.

Lemma 10 constitutes the essence of RPL's DODAG maintenance from the perspective of the considered model and hence reinforces our claims about the potential of the presented techniques for verification of dynamic behaviors of routing protocols. What is equally important, however, is the insight we obtained while proving the lemma. In particular, we identified assumptions under which the lemma holds, found issues in RPL's specification that may require extra treatment by implementers, and gave explicit dependencies between some key configuration parameters, which are not mentioned in the specifications. This knowledge can be employed to improve the reliability of RPL-based systems, both in general and in specific deployment scenarios.

Summary of lessons learned

Overall, in this section, using our model of RPL, we aimed to illustrate how one can verify properties describing dynamic behaviors of a routing protocol. We covered both canonical types of properties that are distinguished in program verification: safety and liveness. Safety properties typically correspond to invariants that have to hold in all or in well-defined states of computations in a system. We thus demonstrated a common approach to checking an invariant, which involves analyzing all possible transitions between global system states and proving that none of them violates the considered invariant. Liveness properties, in turn, often describe various forms of progress that a system is expected to make under specific assumptions. We argued that a good verification heuristic is trying to match a proof of such a property with the way the desired effect it describes spreads in the system, giving a few examples substantiating this argument. We also exemplified that particularly involved properties may require devising custom inductive orders. Finally, we illustrated the iterative nature of a normal verification process, which includes formulating a hypothesis, attempting to verify it, revising its assumptions or the model, and repeating everything, possibly more than once.

Furthermore, while it was not our main goal in this tutorial, we specifically selected the properties serving as our examples so that together they cover a key aspect of RPL's functionality for upward routing: DODAG construction and maintenance. In particular, we formulated invariants bounding rank values and DODAG versions, progress properties for version adoption, and, finally, lemmas formalizing RPL's ability to continuously reselect node ranks and parents in an evolving network, so that globally they correspond to a DODAG. Equally important as the properties themselves is the insight presented while proving them, like the identified assumptions under which they hold or undocumented dependencies between RPL's configuration parameters they entail. In this light, we also tried to discuss practical relevance of individual properties, in particular, where they were not formulated in or even implied by RPL's specification. We recapitulate them shortly. As a final remark, since RPL's upward routing is an application of classic distance-vector routing technique, some of the properties we proved can be applied directly to other routing protocols based on this technique.

In contrast, we did not even aspire to show what one might expect as a "complete repertoire" of verification techniques for dynamic behaviors of routing protocols. We simply believe that no such concept exists. Instead, however, we strove to touch upon all verification aspects that we believe to be relevant, to give sufficiently illustrative examples, and to put everything into perspective so as to demonstrate the behaviors of a key piece of functionality incorporated into routing protocols. We hope that this knowledge will help the readers to start their own verification attempts.

Applying the results in practice

Throughout the tutorial, we gave several examples of how the particular results or insights can be applied in practice. In fact, the presented analyses were actually inspired by real-world problems that we encountered when deploying RPL. The covered techniques were in turn developed to address these problems, which ultimately resulted in fixing existing protocol implementations and adjusting their configurations based on our findings. We are unable to precisely discuss those results here. Nevertheless, we do provide highlights and pointers, so that the interested readers can explore these topics at their discretion.

One example was invalid responses of the two open-source RPL implementations to crashes of border routers (DODAG roots):[8] an entire network either completely collapsed or entered a state of increased resource consumption without actually detecting the failure. It was not clear whether this behavior was due to bugs in the implementations or emergent properties of the protocol itself. Using the techniques covered in

this tutorial, we proved that, by design, RPL is able to handle crashes of border routers or, in general, network partitions, and derived formal conditions that have to be met to this end.[6] In particular, a variant of Lemma 1 was necessary in the proofs, and they involved further safety and global liveness properties. Since some of those conditions had never been explicitly formulated, the implementations simply did not satisfy them, and hence operated incorrectly.

Another example was nonstandard, albeit intuitive route construction criteria taking into account energy consumption, for which the two implementations completely collapsed.[7] Like previously, using the techniques presented here, we derived conditions for provably correct route formation and identified potentially imprecise statements in RPL's specification, which may lead to invalid protocol configurations. In particular, variants of the assumptions used to prove Lemma 8 were one of the results of the process. The insight obtained when deriving these assumptions allowed us to formulate practical guidelines for RPL's implementers, adopters, and designers of future objective functions and route optimization criteria.[7]

What these two examples also illustrate is that the aforementioned risks involved in deploying a routing protocol—even one having mature and widely adopted, and hence seemingly reliable implementations—may indeed be considerable and have profound consequences. They also reinforce the previous claim that guarantees on the behavior of the protocol can help manage such risks, which is more and more relevant with the surging real-world deployments of IoT technologies for controlling physical processes.

Conclusion

To sum up, routing is fundamental technology underlying today's Internet and will likely remain crucial in the emerging IoT. Routing protocols are examples of what is referred to as "complex systems." In particular, they operate on many nodes in a distributed fashion, normally leave some issues open to implementers, rely on external solutions providing side functionalities, involve multiple configuration parameters, and interact with their environment. As a result, predicting their behavior is often nontrivial. At the same time, especially in the context of industrial IoT, where nodes collaboratively control physical processes, overall system dependability requires, among others, precise guarantees on the behavior of the routing protocols employed.

In this tutorial, we illustrated how such guarantees can be derived formally. As a running example of a routing protocol, we adopted RPL, the current de jure standard for low-power wireless networks, which are one of the key communication technologies in IoT. The underlying formalism was in turn based on LTL, which allowed us to focus on actual dynamic behavior (i.e. the evolution) of a system rather than just properties of static system snapshots. After a brief introduction to RPL and LTL, we showed how a routing protocol such as RPL can be modeled in the selected formalism and how such a model can be utilized for verifying the dynamic behavior of the protocol.

In the modeling part, we aimed to illustrate the common problems that one is likely to encounter when devising a formal model of a routing protocol. To avoid excessive complexity, we concentrated on the core, albeit sufficiently broad subset of RPL's DODAG construction and maintenance functionality. We showed how to model processing and communication, abstract algorithms comprising the protocol, and deal with open issues and external dependencies. In our discussion, we emphasized the need for avoiding oversimplifying and overspecifying the model, which would otherwise limit the real-world applicability of the conclusions from verification attempts involving the model.

In the verification part, in turn, we strove to demonstrate how safety and liveness properties—the two types of properties crucial for distributed algorithms and hence routing protocols—can be proved with our model of RPL. To this end, we discussed a few techniques, including invariance checking, contradiction, and induction on various orders. We also exemplified how to approach devising a custom inductive order, which may be necessary for proving more involved properties. Finally, we demonstrated the iterative nature of a typical verification process, including hypothesis formulation, possibly failed proof attempts to identify missing or incorrect assumptions, and gradual hypothesis refinement to make it truly realistic and relevant in practice.

In terms of content, although this was not our main objective, we did cover several key areas of RPL's DODAG construction and maintenance functionality. For example, we proved fundamental bounds on nodes' ranks and DODAG versions in a correctly functioning system, liveness of DODAG version dissemination, and guarantees on the global rank and parent assignment and upkeep. What the tutorial also highlighted as being as important as these results was the insight obtained when producing them. This included assumptions necessary for the proved properties to hold, dependencies between seemingly unrelated configuration parameters, and other issues that are not mentioned explicitly or are simply missing in RPL's specification.

All in all, we believe that this tutorial and its pointers can be of value for both theoretical and practical purposes. From the practical perspective, the presented results can themselves be utilized when extending RPL or designing new distance-vector routing protocols, writing specifications, implementing them, and

configuring for particular deployment scenarios. The aforementioned real-world problems with RPL that inspired our work are a good example. From the theoretical perspective, in turn, the tutorial may guide building new models or retargeting the presented one and proving additional properties. For instance, further use of our methodology, not covered here for the sake of brevity, led to deriving guarantees on RPL's fault tolerance, control traffic behavior, and the impact of asymmetric links; nevertheless, many issues, like the performance of routing adjacency maintenance solutions, downward routing, or multicast, are still open. As our deployment experiences and previous research underlying this tutorial indicate, such knowledge is valuable when aiming at highly dependable systems, as in industrial IoT.

Acknowledgement

The authors would like to thank Agata Janowska for providing early feedback on some parts of this paper. The authors are also grateful to the anonymous reviewers whose comments have helped to improve the quality of this paper.

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This article was supported by the National Science Center (NCN) in Poland under grant no. 2019/33/B/ST6/00448. Its contents also partly draw from our earlier work, supported by the National Center for Research and Development (NCBR) in Poland under grant no. LIDER/434/L-6/14/NCBR/2015.

ORCID iD

Konrad Iwanicki https://orcid.org/0000-0002-5380-6337

References

1. Vasseur JP and Dunkels A. *Interconnecting smart objects with IP: the next Internet*. Burlington, MA: Morgan Kaufmann Publishers, Inc., 2010.
2. Iwanicki K. A distributed systems perspective on industrial IoT. In: *Proceedings of the 2018 IEEE 38th international conference on distributed computing systems (ICDCS)*, Vienna, 2–6 July 2018, pp.1164–1170. New York: IEEE.
3. Winter T, Thubert P, Brandt A, et al. RPL: IPv6 routing protocol for low-power and lossy networks (RFC 6550), 2012, https://www.rfc-editor.org/info/rfc6550
4. Tsiftes N, Eriksson J and Dunkels A. Low-power wireless IPv6 routing with ContikiRPL. In: *Proceedings of the 9th ACM/IEEE international conference on information processing in sensor networks (IPSN'10)*, Stockholm, 12–16 April 2010, pp.406–407. New York: ACM.
5. Ko JG, Eriksson J, Tsiftes N, et al. ContikiRPL and TinyRPL: happy together. In: *Proceedings of the workshop on extending the Internet to low power and Lossy networks (IP + SN'2011)*, Chicago, IL, 12–14 April 2011. New York: ACM.
6. Paszkowska A and Iwanicki K. The IPv6 routing protocol for low-power and lossy networks (RPL) under network partitions. In: *Proceedings of the 2018 international conference on embedded wireless systems and networks (EWSN'18)*, Madrid, 14–16 February 2018, pp.90–101. Junction Publishing/ACM.
7. Paszkowska A and Iwanicki K. On designing provably correct DODAG formation criteria for the IPv6 routing protocol for low-power and lossy networks (RPL). In: *Proceedings of the 2018 14th international conference on distributed computing in sensor systems (DCOSS)*, New York, 18–20 June 2018, pp.43–52. New York: IEEE.
8. Iwanicki K. RNFD: routing-layer detection of DODAG (root) node failures in low-power wireless networks. In: *Proceedings of the 15th ACM/IEEE international conference on information processing in sensor networks (IPSN'16)*, Vienna, 11–14 April 2016, pp.13:1–13:12. New York: IEEE.
9. Paszkowska A and Iwanicki K. Failure handling in RPL implementations: an experimental qualitative study. In: Ammari HM (ed.) *Mission-oriented sensor networks and systems: art and science*, vol. 163 (Studies in systems, decision and control). Cham: Springer International Publishing, 2019, pp.49–95.
10. Pnueli A. The temporal logic of programs. In: *Proceedings of the 18th annual symposium on foundations of computer science (SFCS'1977)*, Providence, RI, 31 October–2 November 1977. New York: IEEE.
11. Ben-Ari M. *Mathematical logic for computer science*. 3rd ed. London: Springer-Verlag, 2012.
12. Cachin C, Guerraoui R and Rodrigues L. *Introduction to reliable and secure distributed programming*. 2nd ed. Berlin; Heidelberg: Springer-Verlag, 2011.
13. Kopetz H and Veríssimo P. Real time and dependability concepts. In: Mullender S (ed.) *Distributed systems*. 2nd ed. New York: ACM Press; Boston, MA: Addison-Wesley Publishing Company, 1993, pp.411–446.
14. Levis P, Lee N, Welsh M, et al. TOSSIM: accurate and scalable simulation of entire TinyOS applications. In: *Proceedings of the 1st international conference on embedded networked sensor systems (SenSys'03)*, Los Angeles, CA, 5–7 November 2003, pp.126–137. New York: ACM.
15. OMNeT + + homepage, http://www.omnetpp.org
16. Osterlind F, Dunkels A, Eriksson J, et al. Cross-level sensor network simulation with COOJA. In: *Proceedings of the 1st international workshop on practical issues in building sensor network applications (SenseApp'06)*, Tampa, FL, 14–16 November 2006, pp.641–648. New York: IEEE.
17. Titzer BL, Lee DK and Palsberg J. Avrora: scalable sensor network simulation with precise timing. In: *Proceedings of the 4th international symposium on information*

processing in sensor networks (IPSN'05), Boise, ID, 15 April 2005. Piscataway, NJ: IEEE Press.
18. Werner-Allen G, Swieskowski P and Welsh M. MoteLab: a wireless sensor network testbed. In: *Proceedings of the 4th international symposium on information processing in sensor networks (IPSN'05)*, Boise, ID, 15 April 2005. Piscataway, NJ: IEEE Press.
19. Doddavenkatappa M, Chan MC and Ananda AL. Indriya: a low-cost, 3D wireless sensor network testbed. In: *Proceedings of the international conference on testbeds and research infrastructures for the development of networks and communities (TridentCom'11)*, Shanghai, China, 17–19 April 2011, pp.302–316. Berlin; Heidelberg: Springer.
20. FIT/IoT-LAB homepage, https://www.iot-lab.info
21. Banaszek M, Dubiel W, Lysiak J, et al. 1KT: a low-cost 1000-node low-power wireless IoT testbed. In: *Proceedings of the 24th ACM international conference on modeling, analysis and simulation of wireless and mobile systems (MSWiM'21)*, Alicante, 22–26 November 2021. New York: ACM.
22. Okola M and Whitehouse K. Unit testing for wireless sensor networks. In: *Proceedings of the 2010 ICSE workshop on software engineering for sensor network applications (SESENA'10)*, Cape Town, South Africa, 3 May 2010, pp.38–43. New York: ACM.
23. Iwanicki K, Horban P, Glazar P, et al. Bringing modern unit testing techniques to sensornets. *ACM T Sensor Network* 2015; 11(2): 25.
24. Astels D. *Test-driven development: a practical guide*. Hoboken, NJ: Prentice Hall, 2003.
25. Klabnik S and Nichols C. *The Rust programming language*. San Francisco, CA: No Starch Press, 2018.
26. Gay D, Levis P, Von Behren R, et al. The nesC language: a holistic approach to networked embedded systems. In: *Proceedings of the ACM SIGPLAN 2003 conference on programming language design and implementation (PLDI'03)*, San Diego, CA, 9–11 June 2003, pp.1–11. New York: ACM.
27. Gay D, Levis P and Culler D. Software design patterns for TinyOS. *ACM T Embed Comput S* 2007; 6(4): 22.
28. Burri N, Flury R, Nellen S, et al. YETI: an Eclipse plugin for TinyOS 2.1. In: *Proceedings of the 7th ACM conference on embedded networked sensor systems (SenSys'09)*, Berkeley, CA, 4–6 November 2009, pp.295–296. New York: ACM.
29. Ramanathan N, Kohler E and Estrin D. Towards a debugging system for sensor networks. *Int J Netw Manag* 2005; 15(4): 223–234.
30. Tolle G and Culler DE. Design of an application-cooperative management system for wireless sensor networks. In: *Proceedings of the 2nd European workshop on wireless sensor networks (EWSN'05)*, Istanbul, 31 January–2 February 2005, pp.121–132. New York: IEEE.
31. Whitehouse K, Tolle G, Taneja J, et al. Marionette: using RPC for interactive development and debugging of wireless embedded networks. In: *Proceedings of the 2006 5th international conference on information processing in sensor networks (IPSN'06)*, Nashville, TN, 19–21 April 2006, pp.416–423. New York: IEEE.
32. Cooprider N, Archer W, Eide E, et al. Efficient memory safety for TinyOS. In: *Proceedings of the ACM embedded networked sensor systems (SenSys'07)*, Sydney, NSW, Australia, 6–9 November 2007, pp.205–218. New York: ACM.
33. Archer W, Levis P and Regehr J. Interface contracts for TinyOS. In: *Proceedings of the 6th international conference on information processing in sensor networks (IPSN'07)*, Cambridge, MA, 25–27 April 2007, pp.158–165. New York: ACM.
34. Romer K and Ma J. PDA: passive distributed assertions for sensor networks. In: *Proceedings of the 2009 international conference on information processing in sensor networks (IPSN'2009)*, San Francisco, CA, 13–16 April 2009, pp.337–348. New York: IEEE.
35. Löscher A, Tsiftes N, Voigt T, et al. Efficient and flexible sensornet checkpointing. In: Krishnamachari B, Murphy AL and Trigoni N (eds) *Wireless sensor networks*. Cham: Springer International Publishing, 2014, pp.50–65.
36. Ribeiro LB, Schlager F and Baunach M. Towards automatic SW integration in dependable embedded systems. In: *Proceedings of the 2020 international conference on embedded wireless systems and networks (EWSN'20)*, Lyon, 17–19 February 2020, pp.85–96. New York: Junction Publishing/ACM.
37. Routing over low power and Lossy networks (roll), https://datatracker.ietf.org/wg/roll
38. Clarke EM, Emerson EA and Sistla AP. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM T Progr Lang Sys* 1986; 8(2): 244–263.
39. Holzmann GJ. The model checker SPIN. *IEEE T Software Eng* 1997; 23(5): 279–295.
40. Bengtsson J, Larsen K, Larsson F, et al. UPPAAL—a tool suite for automatic verification of real-time systems. In: Alur R, Henzinger TA and Sontag ED (eds) *Hybrid systems III: verification and control*. Berlin; Heidelberg: Springer, 1996, pp.232–243.
41. Kwiatkowska M, Norman G and Parker D. PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan G and Qadeer S (eds) *Computer aided verification: 23rd international conference (CAV'2011)*, Snowbird, UT, 14–20 July 2011. Berlin; Heidelberg: Springer, 2011, pp.585–591.
42. Killian C, Anderson JW, Jhala R, et al. Life, death, and the critical transition: finding liveness bugs in systems code. In: *Proceedings of the 4th USENIX conference on networked systems design and implementation (NSDI'07)*, Cambridge, MA, 11–13 April 2007, p.18. Berkeley, CA: USENIX Association.
43. Sasnauskas R, Landsiedel O, Alizai MH, et al. KleeNet: discovering insidious interaction bugs in wireless sensor networks before deployment. In: *Proceedings of the 9th ACM/IEEE international conference on information processing in sensor networks (IPSN'10)*, Stockholm, 12–16 April 2010, pp.186–196. New York: ACM.
44. Li P and Regehr J. T-check: bug finding for sensor networks. In: *Proceedings of the 9th ACM/IEEE international conference on information processing in sensor networks (IPSN'10)*, Stockholm, 12–16 April 2010, pp.174–185. New York: ACM.

45. Mottola L, Voigt T, Österlind F, et al. Anquiro: enabling efficient static verification of sensor network software. In: *Proceedings of the 2010 ICSE workshop on software engineering for sensor network applications (SESENA'10)*, Cape Town, South Africa, 3 May 2010, pp.32–37. New York: ACM.
46. Sobrinho JL. Network routing with path vector protocols: theory and applications. In: *Proceedings of the 2003 conference on applications, technologies, architectures, and protocols for computer communications (SIGCOMM'03)*, Karlsruhe, 25–29 August 2003, pp.49–60. New York: ACM.
47. Hoffmann S and Wanke E. Generic route repair: augmenting wireless ad hoc sensor networks for local connectivity. In: *Proceedings of the 15th international conference on information processing in sensor networks (IPSN'16)*, Vienna, 11–14 April 2016, pp.12:1–12:10. Piscataway, NJ: IEEE Press.
48. Thorup M and Zwick U. Compact routing schemes. In: *Proceedings of the 13th annual ACM symposium on parallel algorithms and architectures (SPAA'01)*, Crete, 4–6 July 2001, pp.1–10. New York: ACM.
49. Burckhardt S, Gotsman A and Yang H. *Understanding eventual consistency*. Technical report MSR-TR-2013-39, 2013. Microsoft Research, https://www.microsoft.com/en-us/research/publication/understanding-eventual-consistency/
50. Helland P. Consistently eventual. *Commun ACM* 2018; 61(8): 50–52.
51. Burckhardt S. Principles of eventual consistency. *Found Trend Program Language* 2014; 1(1–2): 1–150.
52. Vasseur JP, Kim M, Pister K, et al. Routing metrics used for path calculation in low-power and lossy networks (RFC 6551), 2012, https://datatracker.ietf.org/doc/html/rfc6551
53. Hui J and Vasseur JP. The routing protocol for low-power and lossy networks (RPL) option for carrying RPL information in data-plane datagrams (RFC 6553), 2012, https://datatracker.ietf.org/doc/html/rfc6553
54. Levis P, Clausen T, Hui J, et al. The trickle algorithm (RFC 6206), 2011, https://datatracker.ietf.org/doc/html/rfc6206
55. Thubert P. Objective function zero for the routing protocol for low-power and lossy networks (RPL) (RFC 6552), 2012, https://datatracker.ietf.org/doc/html/rfc6552
56. Gnawali O and Levis P. The minimum rank with hysteresis objective function (RFC 6719), 2012, https://datatracker.ietf.org/doc/html/rfc6719
57. Teraoka F, Gogo K, Mitsuya K, et al. Unified layer 2 (L2) abstractions for layer 3 (L3)-driven fast handover (RFC 5184), 2008, https://datatracker.ietf.org/doc/rfc5184/
58. Narten T, Nordmark E, Simpson W, et al. Neighbor discovery for IP version 6 (IPv6) (RFC 4861), 2007, https://datatracker.ietf.org/doc/html/rfc4861
59. Perlman R. Fault-tolerant broadcast of routing information. *Comput Netw* 1983; 7: 395–405.
60. McQuillan JM, Falk G and Richer I. A review of the development and performance of the ARPANET routing algorithm. *IEEE T Commun* 1978; 26(12): 1802–1811.