Proving properties

- `init` is a predicate intended to describe the set of initial states (`pred` denotes a function whose return is a boolean)
- `next` is a predicate that specifies the set of successor states for a given state (relation)
- `invariant` is a function which returns true if for every trace, any element of the trace satisfies `P`

```
traces [SystemState : TYPE,
        init : pred[SystemState],
        next : pred[[SystemState,SystemState]]
       ] : THEORY
BEGIN

  trace?(s:sequence[SystemState]) : bool =
    init(s(0)) AND FORALL (n:nat): next(s(n),s(n+1))

  Trace : TYPE = (trace?)

  invariant(P:pred[SystemState]) : bool =
    FORALL (t:Trace): FORALL (n:nat): P(t(n))
```

Proving properties

The execution of the RPF algorithm is specified as a sequence of network states which starts from an initial state and repeatedly applies a state transition function. For RPF, the initial state models the injection of a packet in the network by the base station. The state transition function models the execution of the algorithm of a generic node x, which is applied recursively to all received packets using an auxiliary `rpf_service` function:

```
rpf_proof_th: THEORY
  BEGIN %-- ... imports omitted

rpf_service(t: nat)(x: node_id)(ns:network_state, g:network_graph, rt:routing_table) : network_state =
  LET scheduled_node = scheduler(t),
      n = size(net_receive_buffer(ns)(scheduled_node))
   IN execute(rpf(scheduled_node)(g,base_station, rt))(n)(ns)

rpf_transition(ns0, ns1: network_state, t: nat)(g: network_graph, rt: routing_table): bool =
  ns1 = rpf_service(scheduled_node)(ns0, g)(base_station, rt)

rpf_trace(seq: sequence[network_state])(g: network_graph, rt: routing_table): bool =
  seq(0) = initial_rpf_state(base_station) AND
   FORALL(t:nat): rpf_transition(seq(t),seq(t+1),t)(base_station,g,rt)

   %-- ... more definitions omitted
  END rpf_proof_th
```

The main property of the RPF algorithm is the following:

Property P: *If the routing table is correct and static, then exactly one copy of the broadcast packet sent by the base station will be delivered to all nodes in the network.*

The formal proof of property **P** is by an induction on the execution traces of RPF, i.e., on sequences that start with the initial state and apply the RPF transition function to generate subsequent states states. Furthermore, the proof makes use of the following properties and constraints, which can be expressed as additional lemmas, or sub-type conditions:

1. the routing table is correct and does not change
2. the network is not partitioned
3. a correct routing table rt exists, which defines a spanning tree rooted at the base station
4. all nodes are guaranteed to be scheduled at least once every N steps of the execution trace, where N is the number of nodes in the network
5. all nodes operate in burst mode: when a node is allowed to transmit, it sends out all packets waiting to be transmitted

Example of proof by induction

Theorem 1 *For every node x, the number of received packets with sender equal to next hop of x is at most one.*

Proof outline. *The proof is given by induction on the number k of hops between the node and the base station on the spanning tree defined by the routing table.*
Base*: $k = 1$. The proof follows from the assumption that the base station injects only one packet.*
Induction*: $k = n + 1$. The path p between the base station and node x is split into a path p' between the base station and next hop of x and an edge between next hop of x and x. The length of p' is n. Hence, the inductive hypothesis holds for the next hop of x, which receives at most one packet. By using Lemma 2, we obtain that x receives at most one packet from the next hop* ◇

Proving properties

SurgeNL protocol. Invariants such as the noLoop property are usually proved by induction on the sequence of generated states. Such induction proofs can be rather complex.

A way to organize proofs is Configuration diagrams.

CONFIGURATION DIAGRAMS: for proving an invariant G, it uses an additional invariant A expressed as a disjunction of predicates $(A = A_1 \vee \cdots \vee A_k)$. Instead of proving G, we prove $G \wedge A$, or, equivalently, $(G \wedge A_1) \vee \cdots \vee (G \wedge A_k)$. The subformulae $G'_i = G \wedge A_i$ are referred to as *configurations*, where i is the index of the configuration.

G= no-Loop-property

A1= base
A2= injected
A3 = preeLoop

The set of configurations can be represented as a labelled graph, called configuration diagram: each vertex corresponds to a configuration; each labelled edge represents a possible transition between configurations, and labels denote transition conditions.

Our starting configuration is called base and it represents the situation in which packet pk^* has not been injected yet, i. e., all receive buffers and all transmission logs of sensor nodes do not contain packet pk^*:

```
base(ns: network_state, pk_star: packet): bool =
  FORALL (i: sensor_id):
         empty?(transmitted(pk_star)(net_log(ns)(i)))
     AND NOT member(pk_star, net_receive_buffer(ns)(i))
```

Configuration
base

In order for this to be a useful configuration in our proof, we need to ensure that it implies he correctness property noLoop_property. This is easy to see, because base confirms that no node contains pk^* in its transmission log, while noLoop_property only requires that for all nodes the set tp of transmitted and logged packets equal to pk^* to be either empty or a singleton. This gives rise to the following lemma. Note that it is lifting the property from network state to the SystemState type in order to make it applicable in the overall invariant proof.

```
s : VAR SystemState

base_implies_noLoop : LEMMA
  base(s`state,pk_star) => noLoop_property(pk_star)(s)
```

We must also prove that the initiality predicate implies conguration base.

```
initially_base : LEMMA
  init_states(s) => base(s`state,pk_star)
```

Assume now that we execute one step of `surge_routingNL`. We have two cases, whether or not a certain sensor `x` is scheduled and will inject the special packet pk^*. These are the transition conditions for configuration `base`. We now consider what will happen in each of these two situations. If pk^* is not injected, other packets will be injected or are forwarded by nodes, but overall the property of `base` does not change, as `base` is only concerned with the special packet pk^*. Hence, we stay in this configuration. If, on the other hand, pk^* is indeed injected, `base` will no longer hold because now there will be a sensor node – `x` – that has transmitted pk^* and thus the first clause of configuration `base` is not valid. Hence, a new configuration has to be introduced; we call it `injected`.

```
injected(ns: network_state, pk_star: packet): bool =
    (EXISTS (j: sensor_id, ts: time):
       (NOT empty?(transmitted(pk_star)(net_log(ns)(j)))) AND
       source_addr(pk_star) = j AND timestamp(pk_star) = ts)
    AND
    (FORALL (i: sensor_id):
       empty?(transmitted(pk_star)(net_log(ns)(i))) OR
       singleton?(transmitted(pk_star)(net_log(ns)(i))))
    AND
    (FORALL (i: {i: sensor_id | singleton?(transmitted(pk_star)(net_log(ns)(i)))}):
       NOT member(pk_star, net_receive_buffer(ns)(i)))
```

Configuration injected

The main part in the development of the configuration diagram is proving that the claimed transitions between the configurations are indeed taken when executing one step of the protocol. For the base configuration we therefore state the following theorem, that expresses that from configuration base we can either stay in base or move to injected:

```
T1: THEOREM
  FORALL (pk_star: packet):
    FORALL (ns: network_state, g: network_graph):
      FORALL (x: sensor_id):
        LET ns_prime = surge_routingNL(x)(ng)(ns) IN
          base(ns, pk_star)
            IMPLIES
          (base(ns_prime, pk_star) OR injected(ns_prime, pk_star))
```

As with base, we ensure that the new configuration satisfies the correctness property noLoop_property, which can easily be seen because the second clause is essentially identical to noLoop_property.

```
injected_implies_noLoop : LEMMA
  injected(s`state,pk_star) => noLoop_property(pk_star)(s)
```

```
preLoop(ns: network_state, pk_star: packet): bool =
  (EXISTS (i: sensor_id, ts: time):
     (NOT empty?(transmitted(pk_star)(net_log(ns)(i)))) AND
       source_addr(pk_star) = i AND timestamp(pk_star) = ts)
  AND
  (FORALL (i: sensor_id):
     (empty?(transmitted(pk_star)(net_log(ns)(i))) OR
       singleton?(transmitted(pk_star)(net_log(ns)(i)))))
  AND
  (EXISTS (j: {j: sensor_id | singleton?(transmitted(pk_star)(net_log(ns)(j)))}):
     member(pk_star, net_receive_buffer(ns)(j)))
```

Configuration preLoop

Note that in configuration `preLoop` there exists one sensor node `x` that both has already transmitted pk^* and has pk^* in the receive buffer. For *SurgeNL* to work correctly, it must be ensured that next time this sensor node is scheduled, it will drop the packets in its receive buffer, and not forward them.

For each configuration:

- prove that the configuration implies noLoop_property
- prove that the identified transitions are always executed when a step of the protocol occurs

Finally, to complete the overall correctness proof of *SurgeNL*, what remains to be shown – besides the fact that `preLoop` satisfies `noLoop_property` – is that the diagram is complete, i. e., in every state of an execution of *SurgeNL* one of the configurations holds. In other words, the disjunction of the configurations is an invariant:

```
diagram_closed(pk_star)(s) : bool =
  LET ns = s`state
  IN
     base(ns,pk_star) OR injected(ns,pk_star) OR preLoop(ns,pk_star)

config_diagram_closed : THEOREM
   invariant(diagram_closed(pk_star))
```

- For security protocols, the model must specify the capabilities of an attacker.

- Several inductively-defined operators are useful.

  (parts) merely returns all the components of a set of messages.
  (analz) models the decryption of past traffic using available keys.
  (synth) models the forging of messages.

The attacker is specified—independently of the protocol—in terms of analz and synth.

- Algebraic laws governing parts, analz and synth have been proved by induction and are invaluable for reasoning about protocols.