Modeling systems using logic

We use the language of PVS

- specification of the system
- animation of the specification
- formal proofs

Although PVS's specification language is based on higher-order logic and features a rich type system, a large subset of it is executable

PVS ground evaluator *PVSio* can be used to animate functional specifications

PVSio enhances PVS language with built-in constructs for string manipulation, floating-point arithmetic and input/output operations

Modelling Wireless Sensor Network Protocols

Generally

- network protocols are described in terms of actions performed by a generic node
- network protocols are described as sequences of steps guarded by control flow conditions
- control fows conditions of network protocols are driven by the value of information items
  - locally stored on nodes (e.g., received packets or local timers), or
  - virtually shared among nodes (e.g., global reference clocks provided by synchronisation protocols).

Modelling Wireless Sensor Network Protocols

- define the generic components: for nodes, network topology, protocol algorithm, ...
- nodes are assigned local services: e.g., packet logger, ...
- the network is assigned global services: e.g., global clock
- function `advance_time` increases a given time by an undetermined amount as follows (where time is used as a synonym for type real):
  `advance_time(t: time): {t1: time | t1 > t}`
- model of computation:
  - synchronous setting where all components take a step simultaneously;
  - asynchronous interleavings of all possible component executions

each component is packaged into a separate PVS model, a theory, and theories can be composed to build more complex components

Abstractions

- number of different versions of the theories can be developed for each component, each theory describing the component at a different level of detail.

- **The most abstract theories provide the declaration of the basic set of mandatory interface functions and types.**
  More detailed theories can be derived from the abstract definitions by specifying the behaviour of functions and by extending types.

- If different versions of a theory provide the same declaration for interface functions and types, they are interchangeable.
  When building the model, we can use the minimal set of details needed for the analysis by importing the appropriate version of the theories.

Nodes

Nodes are numbered starting from `0`

```
%-- definition of types and constants
N: posnat                    %-- symbolic constant
node_id: TYPE = below(N)     %-- a generic node of the network
```

A services `S` is associated to nodes by means of a function:
`[finite_set[node_id] -> S]`.

Depending on the algorithm specification and on the property of interest, services can be installed on a single node, on a group of nodes, or on the entire network.

Network connectivity is modelled with a directed graph without self-edges.
We build type definition on top of directed graphs of the NASA library:
`digraph[node_id]`.

Generic protocol theory

- we use two basic elements: network graph and network state
- we use a function `execute` that: given a network graph, a network state and a protocol function, applies the protocol to the network graph and the network state and returns a new network state

```
protocol_th[network_state: TYPE, network_graph: TYPE]: THEORY BEGIN
  protocol: TYPE = [network_graph -> [network_state -> network_state]]
  execute(ng: network_graph)(p: protocol): [network_state -> network_state] =

     LAMBDA (ns: network_state): p(ng)(ns)
END protocol_th
```

`execute` models the execution of one step of the protocol

Network graph

```
network_graph_th: THEORY
 BEGIN
IMPORTING node_th, digraphs[node_id]

   %-- network_graph: a directed graph without self-edges
   network_graph?(g: digraph[node_id]): bool =
       (FORALL (i: node_id): vert(g)(i)) AND
        (FORALL (i,j: node_id): edges(g)((i,j)) IMPLIES (i /= j))

   network_graph: TYPE = {g: digraph[node_id] | network_graph?(g)}
   topology: TYPE = [node_id -> finite_set[node_id]]


  new_network_graph(tp: topology): network_graph

  %-- instance of network graph
  fully_connected_network_graph: network_graph
     = new_network_graph(LAMBDA (i: node_id): {n: node_id | n /= i})
END network_graph_th
```

Basic network graphs

```
%-- examples of basic network graphs
disconnected_network_graph: network_graph =
  LAMBDA(n:node_id): emptyset[node_id]
linear_network_graph: network_graph =
  LAMBDA(n:node_id): {i: node_id | i = n-1 OR i = n+1 }
fully_connected_network_graph: network_graph =
  LAMBDA(n:node_id): {i: node_id | i/=n }
grid_network_graph(c: posnat): network_graph =
  LAMBDA (i: node_id):
     {n: node_id |
          mod(i, c) > 0 AND (n = i - 1) OR
            mod(i, c) < (c - 1) AND (n = i + 1) OR
              node_id?(i - c) AND (n = i - c) OR
                 node_id?(i + c) AND (n = i + c)}
```

Network state

The network state is described by the set of functions that specify the allocation of services to nodes. For instance, in the following theory, a network state is defined as the collection of two services (receive buffer and log)

```
network_th: THEORY
 BEGIN
   network_state: TYPE =
        [# net_receive_buffer: [node_id -> receive_buffer],
           net_log: [node_id -> log] #]
 END network_th
```

Services

Example of services are:

- packet logger, which stores statistics about sent and received packets,

- receive buffer, which models the buffer where packets sent by other nodes are stored,

- energy consumption, which evaluates the energy spent by nodes,

- routing, which provides the basic definitions for building routing tables, spanning trees and paths between nodes, and

- node scheduler, which gives the sequence of nodes that execute the algorithm (e.g., round robin, or random).

Energy consumption

- network_consumption associates the total used energy to every node
- sender_consumption returns the enery spent when executing send packet
- receiver_consumption returns the energy consumed by a node rcv when the packet is sent by the node snd (different for neighbours/ non neighbours nodes)
- update_network_consumption updates the total energy used by nodes

```
energy_th: THEORY
 BEGIN
   %-- type definition
   energy: TYPE = real
   network_consumption: TYPE = [node_id -> energy]
   %-- interface functions
   sender_consumption: energy
   receiver_consumption(g: network_graph, snd, rcv: node_id): energy
   update_network_consumption(ne: network_consumption,
                              g: network_graph,
                              snd: node_id): network_consumption
 END energy_th
```

General theory for executable specifications

Adding the transition relation

```
execution [State : TYPE] : THEORY
BEGIN
  trans : VAR [State -> State]
  execute(trans)(n:nat) : RECURSIVE [State -> State] =
    LAMBDA (s:State): IF n = 0 THEN s
                      ELSE LET s_new = trans(s) IN
                           execute(trans)(n-1)(s_new)
                      ENDIF
      MEASURE n
END execution
```

The theory takes one type parameter, `State`, and defines a (higher-order) function `execute` that recursively applies `n` steps of a state-transition function `trans`, which is provided as a parameter

`MEASURE` part provides information about termination of the recursion

Traces

Theory that models possible executions of a state transition system as traces.

- Execution traces are modelled as a subtype of possibly infinite sequences of states, namely those that start in an initial state and in which successive elements are connected via the next-state relation
- Using a relational specification of the state transitions allows to express potential non-determinism or the concurrent evolution of the components of a system.

The type `sequence` is predefined in PVS and simply maps natural numbers to corresponding elements of the sequence.

A predicate invariant is declared, which defines what it means for a certain property P to be an invariant of the execution, namely that it holds in every state of any trace.

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`

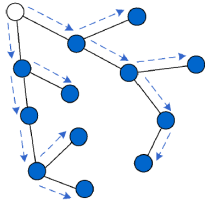
```
traces [SystemState : TYPE,
        init : pred[SystemState],
        next : pred[[SystemState,SystemState]]
       ] : THEORY
BEGIN

  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))

END traces
```

Proving properties

proof by induction on the sequence of system states, as determined by the variable `n`

PVS provides support for different induction schemes, e.g., classical mathematical induction, structural induction on recursive structures, such as lists, trees, graphs or paths....

An example: Flooding protocol



Flooding is a one-to-many routing protocol, in which a dedicated node (the base station) needs to communicate general information to all the nodes of the network.

A simple variant of flooding behaves as follows: whenever a node receives a packet, the packet is forwarded to neighbouring nodes if it is received for the first time, otherwise it is dropped

- define the packet structure
- define the single hop communication primitives
- distinguish between the base station and the other nodes

Packet structure

In the following example, a packet consists of five fields (timestamp, source, sender, destination and payload)

Broadcast address `bcast_addr` is represented with a special constant bcast addr, which is the full set of nodes.

```
packet_th: THEORY
 BEGIN
IMPORTING node_th, time_th
   bcast_addr: finite_set[node_id] = LAMBDA (i: node_id): node_id?(i)
   packet: TYPE =
         [# timestamp: time,
            source_addr: node_id,
            sender_addr: node_id,
            destination_addr: finite_set[node_id],
            payload: finite_sequence[int] #]
 END packet_th
```

Single-hop primitives

Three low level single-hop primitives for communications: inject, forward and drop. Additionally, nodes are also allowed to perform an idle transition.

- **Inject** can be used to send out packets generated by nodes (e.g., packet generated by the application executed on nodes, or control packets generated by the routing service): the function takes a packet as parameter, and sends out such packet.
- **Forward** is suitable to relay packets previously received by nodes (e.g., when multi-hop communication is needed to reach the destination): the function takes a packet as parameter, removes the packet from the receive buffer of the node, and sends out a packet with a sender address automatically updated with the identfier of the sending node.
- **Drop** is used to discard received packets: the function takes a packet as parameter, and removes such packet from the receive buffer of the node.
- **Idle** is useful to update state variables of nodes, such as energy consumption, when no operation on incoming/outgoing packets is performed.

Forward primitive

basic version of the forward primitive

```
network_th_A: THEORY
 BEGIN
   IMPORTING receive_buffer_th
   network_state: TYPE = [# net_receive_buffer: network_receive_buffer #]

   % packet pk is sent out by the forwarder node
   forward(pk: packet)(forwarder: node_id)
          (net: network_state, g: network_graph): network_state =
       LET fw_pk = pk WITH [sender_addr := forwarder],
           nrb0 = update_receivers_buffer(net_receive_buffer(net), g, fw_pk),
           nrb1 = update_sender_buffer(nrb0, pk)
         IN net WITH [net_receive_buffer := nrb1]
 END network_th_A
```

Flooding protocol

Node theory

identification of `base_station` and `sensor_id`; predicates `node_id?`, `base_station?` and `sensor_id?`;

```
node_id_th: THEORY
 BEGIN
   %-- definition of types and constants
   N: posnat                    %-- symbolic constant that specifies the size of the network
   node_id: TYPE = below(N)     %-- a generic node of the network
   base_station: node_id = 0    %-- the base station
   sensor_id: TYPE = {n: node_id | n/=base_station} %-- sensor nodes
   all_nodes : finite_set[node_id] = fullset
   %-- definition of predicates that check the validity of a node indentifier
   node_id?(i: int): bool =  (i>=0 AND i<N)
   base_station?(i: int): bool = (node_id?(i) AND i=base_station)
   sensor_id?(i: int): bool = (node_id?(i) AND i/=base_station)
 END node_id_th
```

Flooding protocol

Protocol theory

```
flooding_th: THEORY
 BEGIN
IMPORTING network_th, log_th
   flooding(x: node_id)(net: network_state, g: network_graph):
         network_state =
       IF empty?(net_receive_buffer(net)(x)) THEN idle(x)(net, g)
       ELSE LET pk = getpacket(net_receive_buffer(net)(x)) IN
              IF empty?(net_log(net)(x)(fw)) THEN forward(pk)(x)(net, g)
              ELSE drop(pk)(x)(net, g)
              ENDIF
       ENDIF
END flooding_th
```

An application

The application consider also mobile nodes. Mobility patterns of nodes are modelled as functions that specify the target location of the mobile nodes through a set of rules. In the case physical locations are not explicitly modelled, nodes' locations are abstracted to node identifiers. (not shown).

```
%-- flooding application
flooding_app: [node_id -> protocol] =
             LAMBDA(x: node_id):
                   IF x = base_station THEN inject(new_packet(x))(x) ELSE flooding(x) ENDIF
%-- flooding execution
flooding_simulation(N_STEPS: nat): network_state =
   LET p = flooding_app,
      ng = grid_network_graph(3),
       s = basic_scheduler(flooding_opt, flooding_sel)
    IN simulate(0, N_STEPS)(p,s,all_nodes,null_mobility)(ng)(empty_network_state)
```

`p` is the protocol and it is equal to `flooding_app`

Function `simulate` is used instead of function `execute` defined above, because we have additional parameters (e.g., the scheduler, the mobility feature etc.)

Simulation output

```
<PVSio> print_state(flooding_execution(3));
EXECUTION STARTS...

>> INITIAL STATE
0:[----]  1:[----]  2:[----]
3:[----]  4:[----]  5:[----]
6:[----]  7:[----]  8:[----]

>> STEP 0, the base station (node 0) gets executed...
0:[----]  1:[#---]  2:[----]
3:[#---]  4:[----]  5:[----]
6:[----]  7:[----]  8:[----]

>> STEP 1, node 1 gets executed...
0:[#---]  1:[----]  2:[#---]
3:[#---]  4:[#---]  5:[----]
6:[----]  7:[----]  8:[----]

>> STEP 2, node 3 gets executed...
0:[##--]  1:[----]  2:[#---]
3:[----]  4:[##--]  5:[----]
6:[#---]  7:[----]  8:[----]

EXECUTION ENDS
==>
```

N_STEPS = 3, the ouput shows the network state and the scheduled node for each

Simulation output

N_STEPS = 3

the ouput shows the network state and the scheduled node for each execution step

the network state consists of the nodes' identifiers and the receive buffers

receive buffers have maximum capacity of four packets

each packet is denoted by the symbol ♯.

Modeling attacks

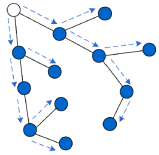
Packet dropping attack. In our framework, we can model the attack through the concept of lossy address. A lossy address is a function that defines which nodes, among those in communication range with the transmitter, will correctly receive the packet. For instance, in the following we show the function declaration for lossy broadcast address:

```
lossy_bcast_addr: {grp: finite_set[node_id] | subset?(grp,
fullset[node_id])}
```

Note that the set of nodes receiving the packet is left unspecified, and, hence, any subset of nodes is possible. If needed, the function can be refined for specifying the mathematical formula describing the precise set of nodes receiving the attack.

Reuse of theories: Reverse Path Forwarding protocol

Reverse Path Forwarding (RPF) is a broadcast routing method which exploits the information contained in the *routing table* to deliver packets generated by a base station to all other nodes in a multi-hop network



With RPF, packets are propagated with the following policy: a node `n` accepts a packet received from node `p` only if `n` believes that `p` is the *best next hop* on the path to the base station, as specified in the routing table.

Under the assumption of a static routing table, the RPF delivers exactly one copy of the broadcast packet to all nodes. If, however, the routing table is dynamic, as is usually the case in real-world deployments, then such guarantees cannot be made for RPF

RPF protocol

```
rpf_th: THEORY
  BEGIN IMPORTING routing_table_th    %-- ... more imports omitted

 rpf(x: node_id)(g:network_graph, base_station:node_id, rt:routing_table)
    (net: network_state): network_state =
   IF empty?(net_receive_buffer(net)(x)) THEN idle(x)(net, g)
   ELSE LET received_pk = getpacket(net_receive_buffer(net)(x))
            sender_addr = sender_addr(received_pk),
            next_hop = next_hop(x, base_station)(g, rt)
        IN IF sender_addr = next_hop
           THEN forward(received_pk)(x)(net, g)
           ELSE drop(received_pk)(x)(net, g)
           ENDIF
   ENDIF

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

RPF protocol

Routing table

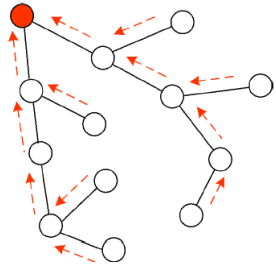
```
routing_table_th: THEORY
BEGIN
IMPORTING network_graph_th, digraphs[node_id]

   routing_table: TYPE = [i: node_id -> [j: node_id -> prewalk[node_id]]]
   routing_table?(rt: routing_table, g: network_graph): bool =
                                      FORALL (i, j: node_id): route?(g, rt(i)(j), i, j)

   valid_route?(g: network_graph, p: prewalk[node_id], i, j: node_id): bool =
       ((i /= j) AND (l(p) > 1) AND path_from?(g, p, i, j))

   valid_routing_table?(rt: routing_table, g: network_graph): bool =
       routing_table?(rt,g) AND FORALL (i, j: node_id): valid_route?(g, rt(i)(j), i, j)

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

RPF protocol

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.

```
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
```

Reuse of theories: Surge protocol



The Surge protocol forms a dynamic spanning tree, rooted at a single node (the base station). Nodes route packets to the root.

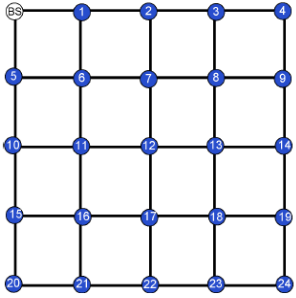
Nodes select a new parent when the link quality falls below a certain threshold. Surge suppresses cycles in the routing by dropping packets that revisit their origin.

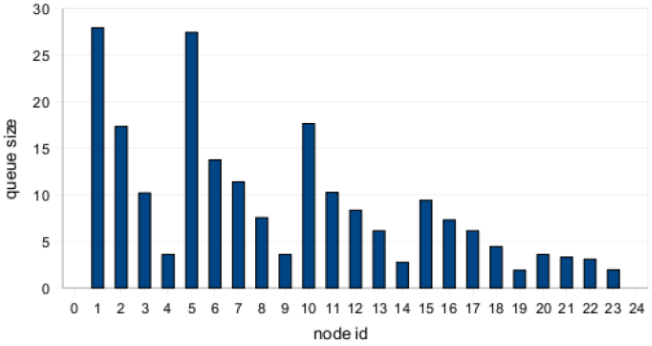
Suppose that we are interested in analysing the forwarding service. The spanning tree service can be assumed correct, i.e., it provides a correct routing table `rt` to the forwarding service.

Surge protocol

```
surge_th: THEORY
 BEGIN
IMPORTING network_th, routing_th

surge(x: node_id)(net: network_state, g: network_graph)
     (base_station: node_id, rt: routing_table): network_state =
    IF empty?(net_receive_buffer(net)(x)) THEN idle(x)(net, g)
    ELSE LET received_pk = getpacket(net_receive_buffer(net)(x)),
             source_addr = source_addr(received_pk),
             sender_addr = sender_addr(received_pk),
             next_hop = next_hop(x,base_station)(g,rt)
          IN IF source_addr /= x
             THEN forward(received_pk
                          WITH [destination_addr := next_hop])(x)(net, g)
             ELSE drop(received_pk)(x)(net, g)
             ENDIF
    ENDIF
```

Surge protocol



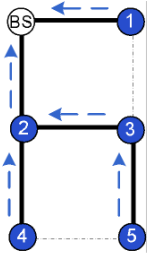


Results for a 5x5 grid network.

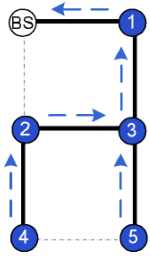
The protocol has been simulated several times and for different number of steps. For high number of steps, the queue size almost stabilised.

As expected, the maximum number of packets in the receive queue is larger for nodes closer to the base station, because they have to relay packets for more nodes.

Robustness to topology changes



Spanning tree sp_A



Spanning tree sp_B

We were able to detect a potential problem of infinite loops of routed packets in the algorithm specification.

There are situations in which a packet may travel indefinitely in the network, because the routing table may change in response to topology changes.

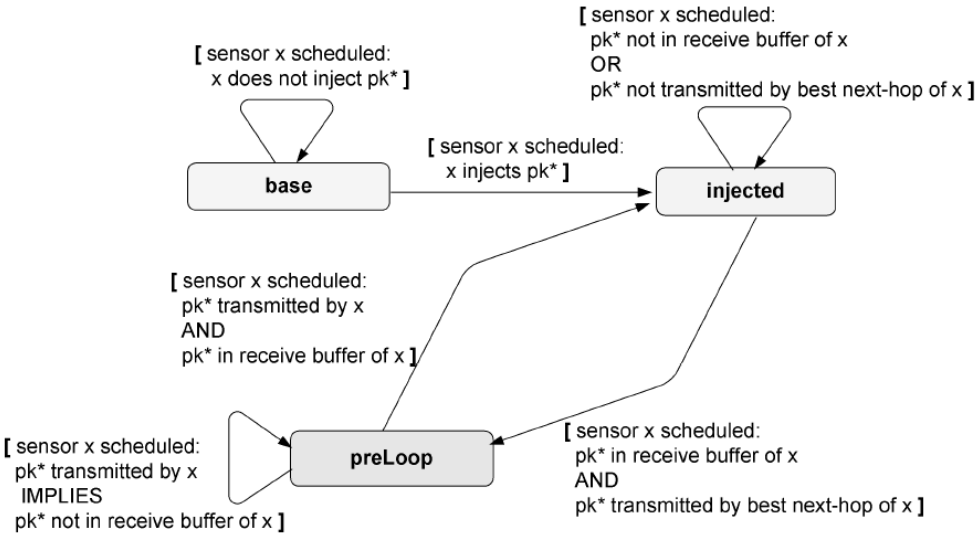
Surge suppresses cycles in the routing by dropping packets that revisit their origin.

A revised version of Surge

A new version of the protocol has been suggested:

- SurgeNL uses bursty transmissions, i.e., each node always transmits all packets ready to be sent, while Surge does not give any constraint on this aspect;
- SurgeNL guarantees that *each node always injects a new packet whenever it performs a transmission*, while Surge sends only the packets that are stored in the receive buffer (such packets are those received from other nodes);
- SurgeNL detects routing loops by inspecting the source address of all packets in the receive buffer, while Surge, on the other hand, inspects only the source address of the packet to be transmitted.

A revised version of Surge



A revised version of Surge

Configuration `base`

```
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))
```

A revised version of Surge

Configuration `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)))
```

A revised version of Surge

Configuration `preloop`

```
%-- configuration preLoop
  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)))
```

Surge protocol

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))
```

Formal specifications

- the specifications are intuitive for designers and practitioners
- the level of detail of the specification can be seamlessly adapted: analysts can construct high-level specifications of the protocol, and then refine such specifications for taking into account specific situations (e.g., networks with mobile nodes) and specific properties of interest
- modular and reusable theories
- attacks (as well as non-malicious faults) can be simply modeled and integrated into the specification
- the same formal specifications can be used both in the theorem prover for proving key properties of the protocol, and in the environment for simulation

Cryptographic protocols

Theorem provers have been used for stating and proving properties of cryptographic protocols: Isabelle/HOL, Coq, ...

The analysis of the Needham-Schroeder protocol which uses public-key encryption was studied in
Paulson L., The inductive approach to verifying cryptographic protocols, Journal of Computer Security, 1998

From Paulson's paper:
Informal arguments that cryptographic protocols are secure can be made rigorous using inductive definitions. The approach is based on ordinary predicate calculus and copes with infinite-state systems. Proofs are generated using Isabelle/HOL. The human effort required to analyze a protocol can be as little as a week or two, yielding a proof script that takes a few minutes to run.