Modeling and verification certain classes of systems

extensions of finite state-transition systems

- Model Checking Real-Time Systems
  TCTL formulae, a natural extension of CTL
- Model Checking Security Protocols
- Model Checking Probabilistic Systems

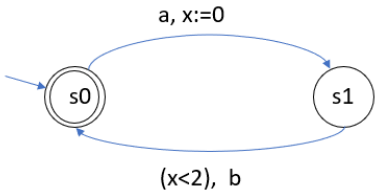
Timed Automata

- A timed automaton consists of an automaton (states and transitions) and real-valued variables for measuring time between transitions, named Clocks
- Clocks progress synchronously with time (time domain R^+), and can be reset by transitions
- a transition consists of a guard, an action and a reset of clocks
- invariants can be added to states
- The operational semantics of a timed automaton is an (infinite-state) timed transition system.

Rajeev Alur and David L. Dill.
A theory of timed automata, Theoretical Computer Science, 126, 2, 1994

An example

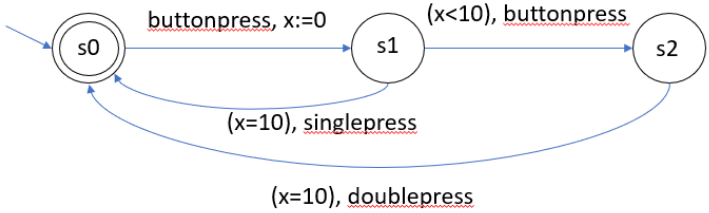
This timed automaton has one clock: x



This automaton moves from initial state $s0$ to $s1$ by executing action a. The clock x is set to 0 along this transition.
While in state $s1$, the clock x shows the time elapsed since the occurrence of the last a.
The transition from $s1$ to $s0$ is enabled only if this value is less than 2.
The the automaton moves back to state $s0$ and the cycle is repeated.

Another simple example

This timed automaton has one clock: x

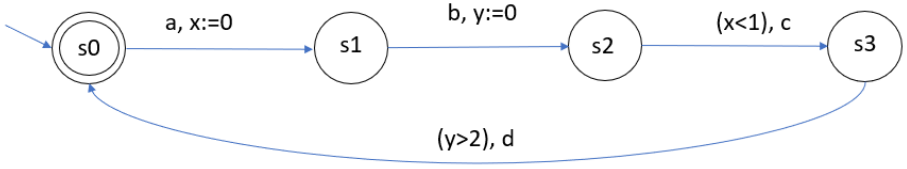


This automaton leaves the initial state *s*0 by executing action *buttonpress*, and reset the clock *x* to 0. If action *buttonpress* is executed again before 10, the automaton moves to state *s*2, and, when the clock reaches 10, recognises a double click of the button. Otherwise, being in state *s*1, when the clock reaches 10, the automaton recognises a single click of the button.

Another example

The timed automaton has two clocks: x and y.
The automaton cycles among the states $s0$, $s1$, $s2$, $s3$

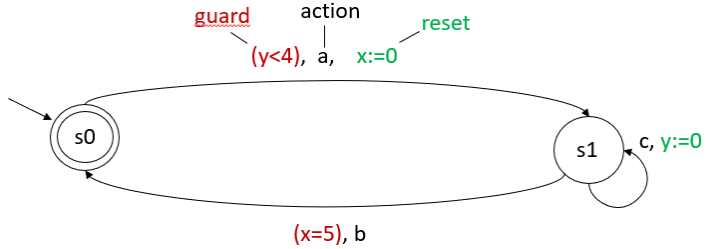
Clock x is set to 0 each time it moves from $s0$ to $s1$ executing a.
c happens within time 1 from the preceeding a
clock y is set while executing b
the delay between b and the following d is greater than 2



Timed Automata

x,y: clock

transition relation: $s \xrightarrow{action,time} s'$



$$s_0 \xrightarrow[3.2]{a} s_1 \xrightarrow[5.1]{c} s_1 \xrightarrow[8.2]{b} s_0 \cdots$$

value of x:	0	0	1.9	5	$\cdots$
value of y:	0	3.2	0	3.1	$\cdots$

Timed Automata

A transition is labelled with one action, a constraint (guard) and the reset of zero or more clocks (a, g, r). A reset is a clock to be set to zero.

The *state* of a timed automaton is given by the location and the values of the clocks at a given time.
State: $(location, x = v, y = w)$ with $v, w \in R^+$

Execution trace:

$$(s_0, x = 0, y = 0) \xrightarrow{a} (s_1, x = 0, y = 3.2) \xrightarrow{c}$$
$$(s_1, x = 1.9, y = 0) \xrightarrow{b} (s_0, x = 5, y = 3.1) \cdots$$

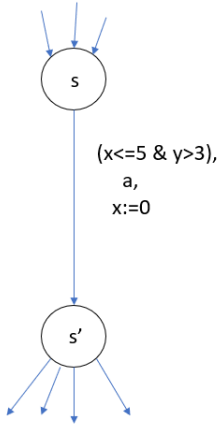
A timed automaton models a system operating in a number of distinct *modes*. Each mode is represented by a location. Mode changes occur as a consequence of the execution of actions.

While the system remains in a given location, progress of time is reflected by the values of the clocks, whose values increase all at the same rate.

Adding invariants

Let us consider the following transition of a timed automaton.

Until *a* does not occur, time alone flows



Clocks: x, y

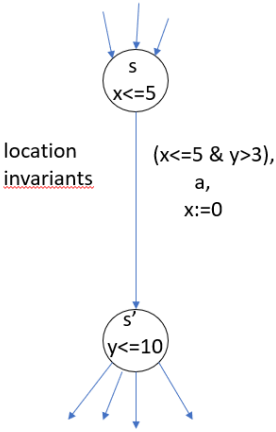
Transitions:

(s, x=2.4, y=3.1415) —a→ (s', x=0, y=3.1415)

(s, x=2.4, y=3.1415) —wait(1,1)→ (s, x=3.5, y=4.2415)

Adding invariants

Invariants ensure progress



Clocks: x, y

Transitions:

(s, x=2.4, y=3.1415) —wait(3.2)→ ✗

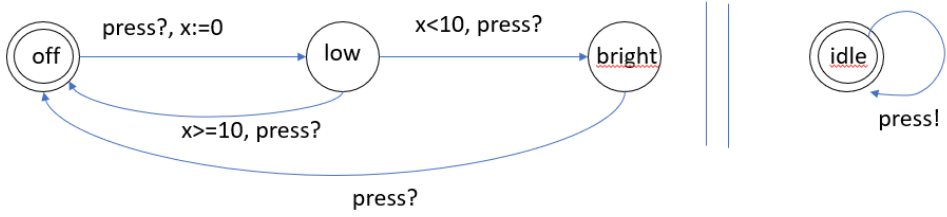
(s, x=2.4, y=3.1415) —wait(1,1)→ (s, x=3.5, y=4.2415)

Network of timed automata

Complementary actions: two actions of the same name *a*? and *a*! are said to be complementary, or matching.

In a network of timed automata, the complementary actions are synchronisation actions and become the network internal action, denoted by τ.

Example: a simple lamp



Semantics

Definition (timed automaton)

A *timed automaton* is a tuple $\mathbf{A} = (\mathcal{L}, l_0, \mathcal{C}, E, \Sigma, \mathcal{I})$, where

- $\mathcal{L}$ is a finite set of *locations*;
- $l_0 \in \mathcal{L}$ is the *initial* location;
- $\mathcal{C}$ is a finite set of *clocks*;
- Σ is the set of actions;
- $E \subseteq \mathcal{L} \times \Sigma \times B(\mathcal{C}) \times 2^{\mathcal{C}} \times \mathcal{L}$ is the set of *edges* between locations with an action, on the set of clocks and a set of clocks to be reset;
- $\mathcal{I} : \mathcal{L} \to B(\mathcal{C})$ assigns invariants to locations.

Semantics

Let $\mathbb{R}$ be the set of non-negative reals, and u be a clock valuation: $u \subseteq \mathcal{C} \times \mathbb{R}$.

Definition (Semantics of a timed automaton)

The semantics of a timed automaton is a labelled transition system $\langle S, s_0, \rightarrow \rangle$ where $S \subseteq L \times \mathbb{R}^{\mathcal{C}}$ is the set of states, $s_0 = (l_0, u_0)$ is the initial state, and $\rightarrow$: $S \times (\Sigma \cup \mathbb{R}) \times S$ is a *transition relation* such that:

- $(l, u) \xrightarrow{d} (l, u + d)$ *if* $\forall d' : d' \in [0, d] \Rightarrow u + d' \in \mathcal{I}(l)$, and
- $(l, u) \xrightarrow{a} (l', u')$ *if* there exists $e = (l, a, g, r, l') \in E$ such that: $u \in g$, $u' = [r \mapsto 0]u$, and $u' \in \mathcal{I}(l')$

  where $u + d$ maps each clock $x \in \mathcal{C}$ to the value $u(x) + d$, and $[r \mapsto 0]u$ denotes the clock valuation which maps each clock in r to 0 and agrees with u over $\mathcal{C} - r$.

Semantics

Definition (Network of timed automata)

A *network* of timed automata consists of n timed automata over the same set of actions and clocks, $\mathbf{A}_i = (L_i, l_i^0, C, \Sigma, E_i, \mathcal{I}_i)$, $i = 1, \cdots, n$.

Let $\bar{l} = (l_1, \cdots, l_n)$ be a location vector

Let $\mathcal{I}(\bar{l}) = \wedge_{i=1,\cdots,n} \ \mathcal{I}_i(l_i)$ be the composition of the invariant functions over location vectors

Let $\bar{l}\,[l_i'/l_i]$ to denote the vector where the i-th element l_i of $\bar{l}$ is replaced by l_i'

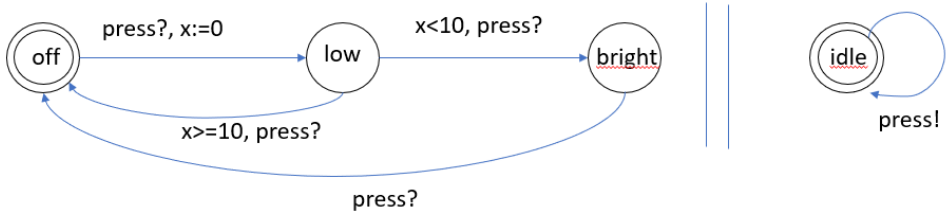
Semantics

Definition (Semantics of a network of timed automata)

The semantics of a network of timed automata is defined as a transition system $(\mathbf{S}, s_0, \rightarrow)$, where: $(\mathbf{S} = \mathcal{L}_1 \times \cdots \mathcal{L}_n) \times \mathcal{R}^c$ is the set of states, $s_0 = (\bar{l}_0, u_0)$ is the initial state and $\rightarrow \subseteq S \times S$ is the transition relation defined by:

- $(\bar{l}, u) \xrightarrow{d} (\bar{l}, u + d)$ *if* $\forall d' \in [0, d] \rightarrow (u + d') \in \mathcal{I}(\bar{l})$.
- $(\bar{l}, u) \xrightarrow{a} (\bar{l}\,[l_i'/l_i], u')$ *if* there exists $l_i \xrightarrow{a,g,r} l_i'$ such that $u' = [r \rightarrow 0]u$ and $u' \in \mathcal{I}(\bar{l}\,[l_i'/l_i])$.
- $(\bar{l}, u) \xrightarrow{\tau} (\bar{l}\,[l_j'/l_j,\ l_i'/l_i], u')$ *if there exists* $l_i \xrightarrow{c?,g_i,r_i} l_i'$ and $l_j \xrightarrow{c!,g_j,r_j} l_j'$ such that $u \in (g_i \wedge g_j)$, $u'[r_i \cup r_j \mapsto 0]u$ and $u' \in \mathcal{I}(\bar{l}\,[l_j'/l_j,\ l_i'/l_i])$

An example



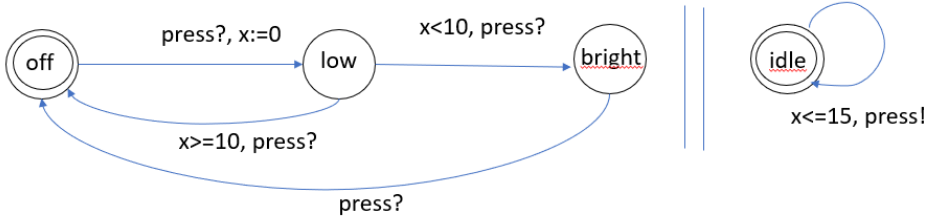
clock: x

Some possible executions ($\xrightarrow{d}$ are not shown)

$((off, idle), 0) \xrightarrow{\tau} ((low, idle), 0) \xrightarrow{\tau} ((bright, idle), 5.1) \xrightarrow{\tau} ((off, idle), 7) \xrightarrow{\tau} (low, 0) \quad \cdots\cdots$

$((off, idle), 0) \xrightarrow{\tau} ((low, idle), 0) \xrightarrow{\tau} ((off, idle), 22.5) \xrightarrow{\tau} ((low, idle), 0) \cdots\cdots$

Behaviour with a guard



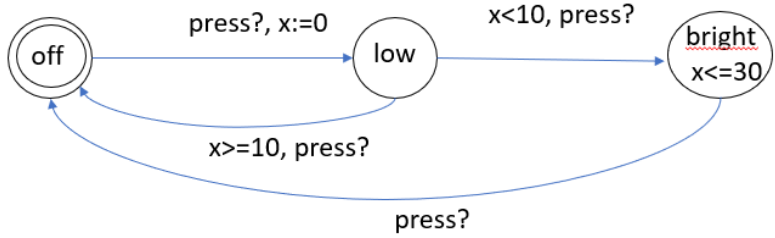
Some possible executions

$((off, idle), 0) \xrightarrow{\tau} ((low, idle), 0) \xrightarrow{\tau} ((bright, idle), 5.1) \xrightarrow{\tau} ((off, idle), 7) \xrightarrow{\tau} (low, 0) \cdots\cdots$

$((off, idle), 0) \xrightarrow{\tau} ((low, idle), 0) \xrightarrow{\tau} ((off, idle), 12.5) \xrightarrow{\tau} ((low, idle), 0) \cdots\cdots$

The execution $((off, idle), 0) \xrightarrow{\tau} ((low, idle), 0) \xrightarrow{\tau} ((off, idle), 22.5)$ is NOT POSSIBLE because the transition must satisfy the guard $x <= 15$

Behaviour with an invariant



clock: x

$(off, 0) \xrightarrow{\tau} (low, 0) \xrightarrow{\tau} (off, 32) \xrightarrow{\tau} (low, 0) \cdots\cdots$

$(off, 0) \xrightarrow{\tau} (low, 0) \xrightarrow{\tau} (bright, 5.1) \xrightarrow{\tau} (off, 7) \xrightarrow{\tau} (low, 0) \cdots\cdots$

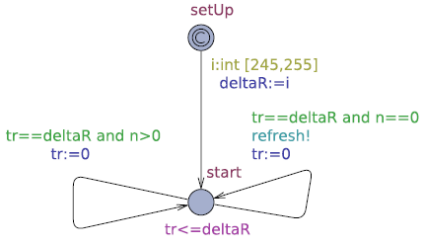
$(off, 0) \xrightarrow{\tau} (low, 0) \xrightarrow{\tau} (bright, 5.1) \xrightarrow{\tau} (off, 32) \xrightarrow{\tau}$ is NOT POSSIBLE because location *bright* must satisfy the invariant

Software tools

- Timed automata are a formalism for model checking real-time systems
- A model checker tool for the analysis of timed automata is UPPAAL
  https://uppaal.org/
  G. Behrmann, A. David, and K. G. Larsen, A tutorial on UPPAAL 4.0, 2006.
  https://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf
- UPPAAL extends timed automata with integer variables, structured data types (e.g., arrays and records), user defined functions and channel synchronisation (e.g., broadcast channels, committed and urgent locations)
- simulation
- model checking of a simplified version of Timed CTL (TCTL) formulae

Example: a timed automaton in UPPAAL

The timed automaton operates on one clock `tr` and a variable `n`. When the clock is equal to the constant `deltaR`, an out edge is executed (`tr <= deltaR` is the state invariant). If variable `n` equals to 0, the TA sends a `refresh` message and resets the clock. Otherwise (`n > 0`), only resets the clock.



The TA models the automatic refresh actions performed by a GUI every `deltaR` time units, in absence of user actions (n==0). Initially, a random value in the interval $[245, 255]$ms is assigned to `deltaR` (4 Hertz)

Locations in UPPAAL

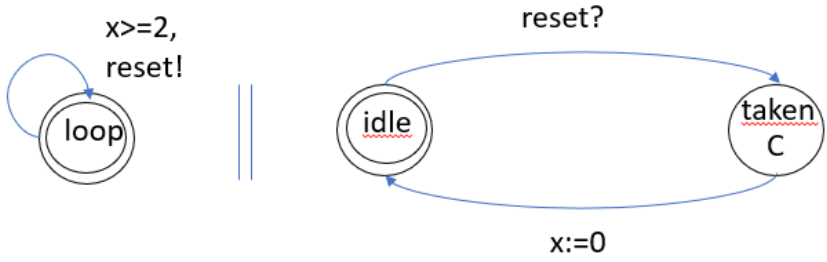
- normal location with or without invariants
- Urgent locations
  equivalent to add an extra clock y that is reset on all incoming edges, and having an invariant $y <= 0$ on the location
- Committed locations
  a state is comitted if any of the locations in the state is committed.
  A committed state cannot delay and the next transition must involve an outgoing edge of at least one of the committed locations
  (more restrictive than urgent locations)

Clocks and guards

- invariants
  an invariant is a conjunction of conditions of the form:
  $x < expr$ or $x <= expr$,
  where x is a clock and *expr* evaluates to an integer.
  invariants must be side-effect free

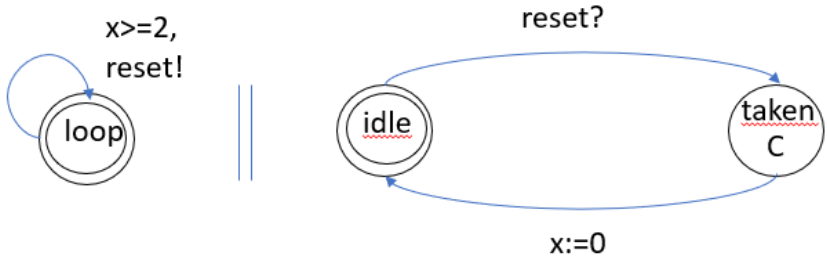
- guards
  expression that evaluates to boolean; clocks and clock differences compared to integer expressions.
  guards over clocks are conjunctions.
  only clocks, integer variables and constants are referenced.
  guards must be side-effect free

Example 1



Possible behaviours?

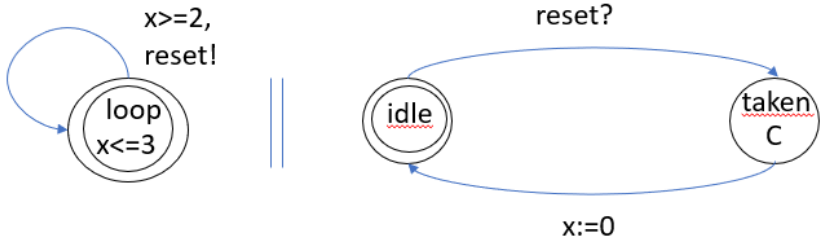
Example 1



A possible execution:
$((loop, idle), 0) \xrightarrow{\tau} ((loop, taken), 2) \xrightarrow{\tau} ((loop, idle), 0) \xrightarrow{\tau} ((loop, taken), 8) \xrightarrow{\tau} ((loop, idle), 0) \cdots\cdots$

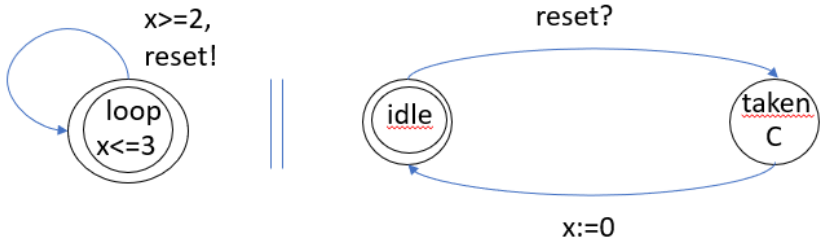
Starting from the initial state with $x = 0$, after a delay $>= 2$ the net can move into state $(loop, taken)$. Since the state is committed, the outgoing edge from the committed location must be executed, ad the clock is reset to 0, and we move to the initial state.
All reset of x will happen when $x \geq 2$.

Example 2



Possible behaviours?

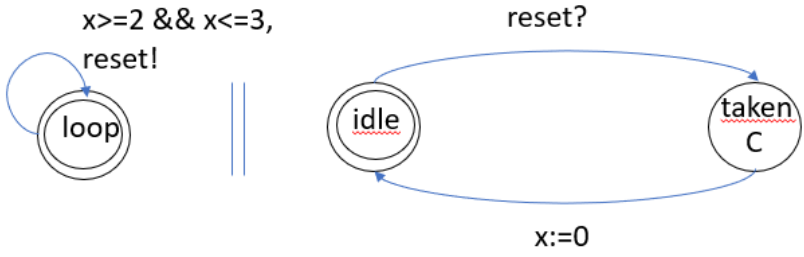
Example 2



A possible execution:
$((loop, idle), 0) \xrightarrow{\tau} ((loop, taken), 2) \xrightarrow{\tau} ((loop, idle), 0) \xrightarrow{\tau} ((loop, taken), 2.8) \xrightarrow{\tau}$
$((loop, idle), 0) \xrightarrow{\tau} ((loop, taken), 1) \cdots\cdots$

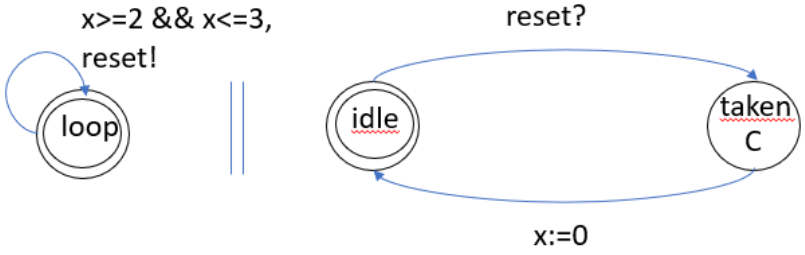
The system is not allowed to stay in state $(loop, idle)$ for more than 3 time units.
$((loop, idle), 0) \xrightarrow{\tau} ((loop, taken), 8)$ is not possible.
The transition is taken after a delay between 2 and 3. Clock x has 3 as upper bound.

Example 3



Possible behaviours?

Example 3



A possible execution:
$((loop, idle), 0) \xrightarrow{\tau} ((loop, taken), 2) \xrightarrow{\tau} ((loop, idle), 0) \xrightarrow{\tau} ((loop, taken), 2.8) \cdots\cdots$

Another execution:
$((loop, idle), 0) \xrightarrow{\tau} ((loop, taken), 2)$

Same behaviour as before, but deadlock may also occur.

Model checking

E - exists a path (E in UPPAAL).
A - for all paths (A in UPPAAL).
G - all states in a path ([] in UPPAAL).
F - some state in a path (<> in UPPAAL).

A[]p, A<>p, E<>p, E[]p and p –> q

where p is a state formula
p::= a.l | g | p and p | p or p | not p | p imply p

a.l: automaton a is in location l
g: conditions on variables and clocks.

Model checking

Properties that can be directly checked using the UPPAAL: reachability, liveness, safety formulae.

`deadlock` is a special state formula. A state satisfies `d`eadlock if there are no outgoing actions. This formula can be used to check if the system is deadlock free.

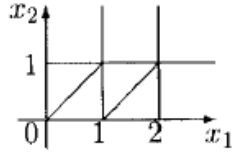
A[] not deadlock
is a special case in UPPAAL for proving that for each reachable state, there exists at least one outgoing edge.

Reachability analysis

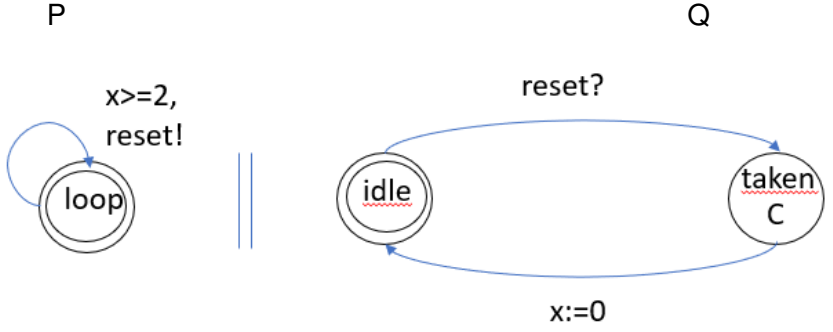
- Only continuous variables are timers
- Invariants and Guards: x<const, x<=const
- Actions: x:=0
- can express lower and upper bounds on delays
- can partition the state space into a finite set of regions

Let C_i the greatest constant the clock variable x_i is compared with. The regions are determined according to C_i.

Consider two clocks x_1 and x_2 with $C_1 = 2$ and $C_2 = 1$, we have the following regions:



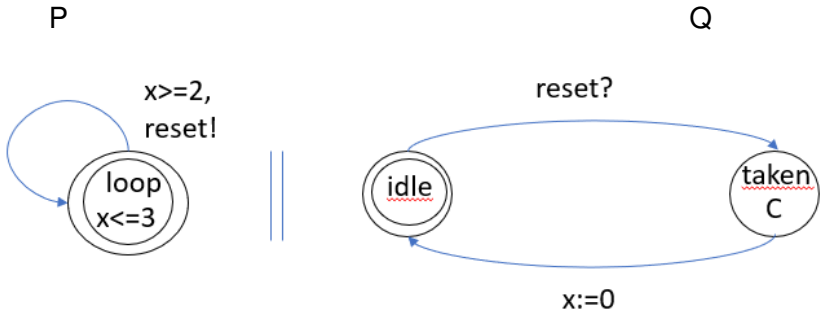
Example 1



A[] Q.taken imply x>=2
in all states in which Q is in location *taken*, the clock satisfies $x >= 2$
when x is reset, x is $>= 2$

E<> Q.idle and x>3
exists a state in which Q is in location *idle* and the clock satisfies $x > 3$
it is possible to delay more than 3 units before the reset

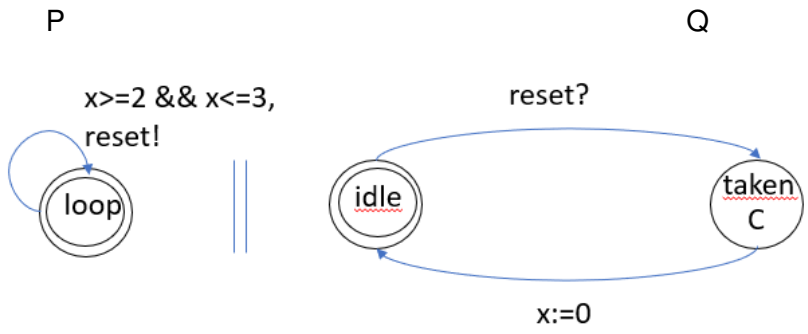
Example 2



A[] Q.taken imply (x>=2 and x<=3)
the clock is reset after a delay between 2 and 3

E<> Q.idle and x>3
FALSE

Example 3



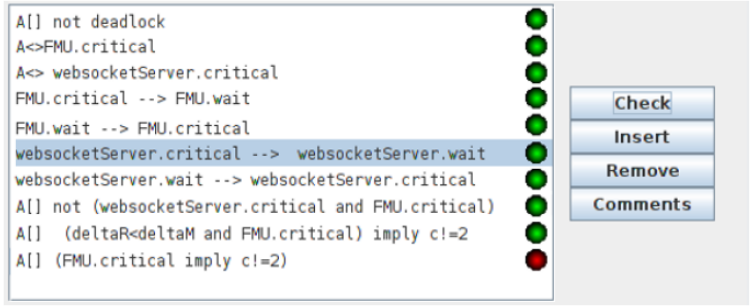
A[] not deadlock
FALSE

A[] Q.idle imply x<=3
FALSE

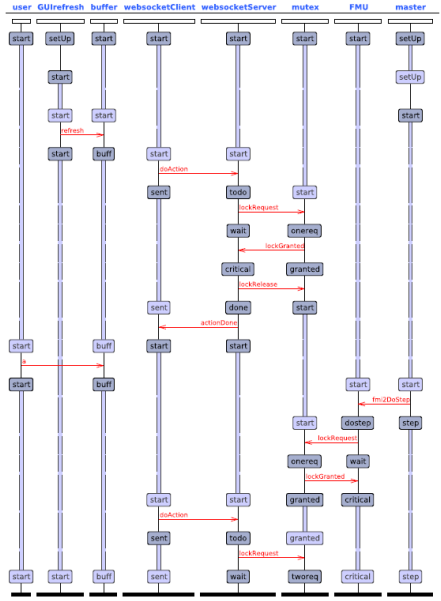
the system has no progress condition
P can delay in state *loop* for more than 3 time units.

The tool

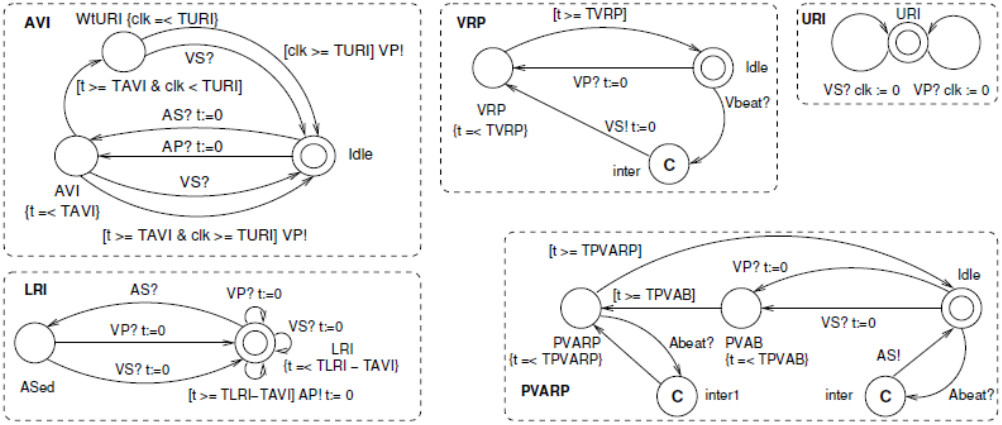
A[] (deltaR < deltaM and FMU.critical) imply c!=2
proves a property under specific timing constraints



Simulation


user
GUIrefresh
buffer
websocketClient
websocketServer
mutex
FMU
master
start
setUp
refresh
doAction
sent
todo
lockRequest
wait
onereq
lockGranted
critical
granted
lockRelease
done
actionDone
buff
a
fmi2DoStep
dostep
step
tworeq

A network modeling the behavior of a pacemaker



A network modeling the behavior of a pacemaker

Local clocks. Each automaton enforces some constraint on the intervals between pairs of events, using clocks named t (this name is local to each automaton).

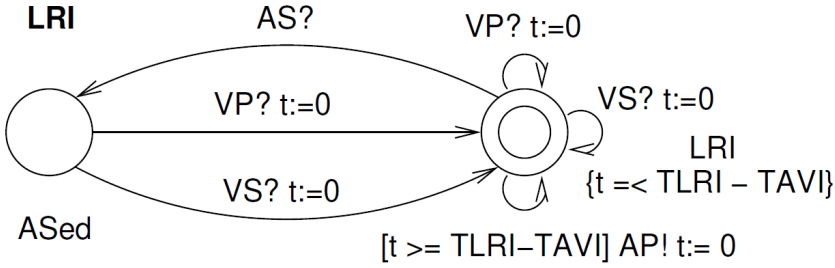
Global clocks. clk is a global clock and it is accessed both by the URI and the AVI automata.

Square brackets enclose guards and curly brackets enclose state invariants. Two concentric circles denote the initial location. Letter *C* denotes committed locations, meant to model an immediate transition.

Actions *VS*?, *VP*!, *AS*? and *AP*! are synchronisation actions towards the model of the heart (not shown here).

LRI automaton

The LRI automaton keeps the heart rate above a minimum value, defined by the Lower Rate Interval (TLRI).



LRI automaton

The automaton starts in the LRI location. Upon a ventricular event (VP? or VS?), clock t is reset. Upon an atrial sense event (AS?), the automaton waits in location ASed until a ventricular event occurs, which causes the automaton to return to the LRI location, resetting t.

Since an atrial event must occur within $TLRI - TAVI$ seconds after the last ventricular event, an atrial pulse (AP!) must occur if the automaton remains in LRI for a longer interval.

This is specified by the invariant $\{t \in TLRI - TAVI\}$ on location LRI and by the edge labeled with the guard $[t > TLRI - TAVI]$, the action AP!, and the reset of t. This models the issue of an atrial pacing pulse by the pacemaker.

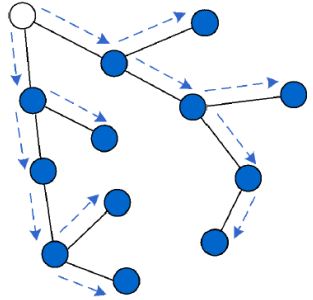
Modeling attacks: an example

- modelling a simple WSN protocol
- modelling of attacks and injection of attacks
- analysis of the protocol behaviour in absence and in case of attacks

Model based design of security.

Modelling a simple WSN protocol

Flooding. 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.



As an example, flooding can be applied for dynamic route discovery.

Modelling a simple protocol

A simple version of flooding behaves as follows: whenever a network node receives a message, it is forwarded to all its neighbors only if it has not already been forwarded; otherwise, it is dropped. Moreover, nodes drop old messages also, when received.

An interesting property of the flooding protocol is the following:

Property P: every node receives all the messages sent by the base station, and every message that was received is then forwarded only once.

Protocol specification

- designers and practitioners generally describe communication protocols in terms of actions performed by a generic networked node
  For example, *the first time a node receives a message (packet), it is re-broadcast*.

- protocols are described as sequences of steps guarded by control flow conditions
  generally, control flows 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).

Protocol specification

- in communication protocols for wireless sensor networks, communication channels must be carefully modelled, because they may affect how information flows in the network, and because they are sensible to several factors, such as physical distance between nodes, and number of concurrent communications
- we must define abstractions that retain the information needed for the analysis

Protocol specification

old messages: messages uniquely identified by their timestamp

originator, data?: not needed for the analysis. A message is abstracted to its timestamp

each node receive all messages sent by the Base station: timestamp

each node forwards a message only once: for each node, log file of forwarded messages

Base station sends messages in sequence with a delay of 1 time unit.
Other nodes: no delay model for the reception and forwarding of messages

concurrent execution: for each node, buffer of received messages;

Modelling elements in UPPAAL

- template system
  Basically, network nodes have only one main state from which a set of outgoing transitions starts. Such transitions account for the reception/transmission of packets from/to other network nodes.
  the node process is named `node` and has `node_t` as parameter. Every node ha a differnt id. All instances of the template `node` ranging over its parameter are generated.

- broadcast communication
  Network nodes are connected through broadcast channels (`broadcast chan c`).
  UPPAAL *broadcast* channel allows one sender to synchronise with an arbitrary number of recievers. Receivers that can synchronise, must so. Broadcast are not blocking, if there are no receiver the sender can execute the send.

Modelling elements in UPPAAL

- A clock `clk` is used to implement the process of generation of new messages after a given delay.
- functions
  auxiliary function are defined to improve readability of the specification
- global and local data structures

Protocol specification

- NODES: number of nodes. Nodes are numbered starting from 0;
- `MSGS` represents the number of messages sent by the Base station; messages are numbered starting by 1 to *MSG*;
- `DIM` is length of array local to nodes

- global structures
  - `chnl` array: broadcast communication channel (indexed by node id);
  - `bmessages` array: stores the messages broadcasted by the nodes at a protocol step, (indexed by node id);

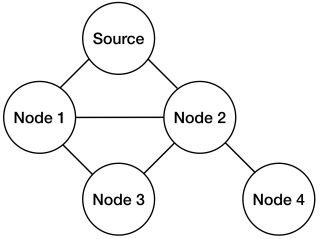
Protocol specification

- structures local to a node
  - `TS`: the most recent timestamp of received messages;
  - `buffer` array: a FIFO buffer that stores messages input to the node waiting to be forwarded, with *n* the current number of elements; the message to be sent is in `buffer[0]`.
  - `logger` array: a queue which stores the already broadcasted messages, with *m* the current number of elements. Tis array is used to check the *Property P*.

- node_t $\in$ [0, NODES-1], checker_t $\in$ [0, MSGS-1], size_t $\in$ [0, DIM-1].

The network

We refer to a WSN made up of five nodes i) one source node; and ii) four relay nodes.



We model the topology of the network by a function
$neighbour : (NODES \times NODES) \rightarrow Bool$

$\forall i, j \in NODES, neighbour(i, j) = true$ if j is in the communication range of i and vice-versa; *false* otherwise

The network

We assume the communication range between nodes is one hop. The function `receive(id, i)` of relay nodes is tailored on such network parameters, and returns `true` if the nodes `id` and `i` are one-hop neighbours, `false` otherwise.

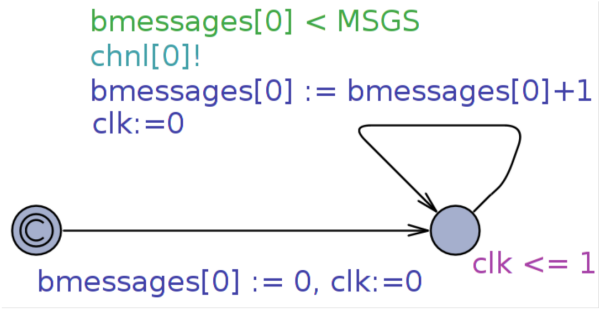
As an example, `receive(4, 2)` returns `true`, since Node 2 is a one-hop neighbour of Node 4.

Conversely, `receive(4, 3)` returns `false`, since Node 3 is two-hops far away from Node 4.

Global declarations

```
const int NODES = 5;
typedef int[0,NODES-1] node_t;
typedef int timestamp;
chan chnl [node_t];
timestamp bmessages[node_t];
```

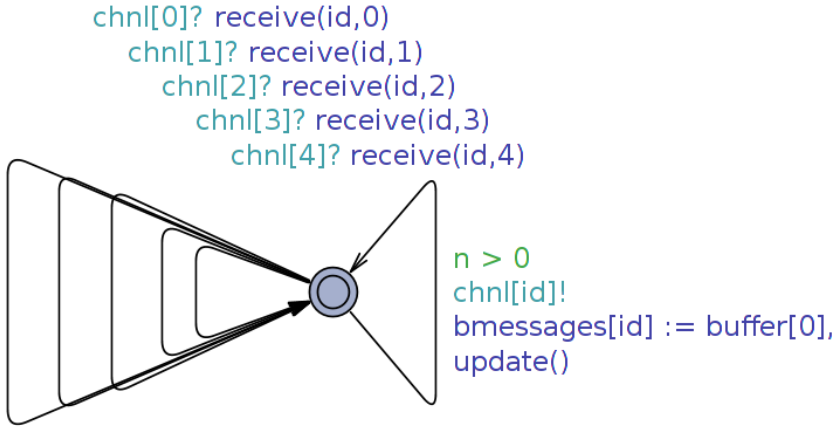
Abstract model of the source node.



The source node broadcasts a brand new message at most every 1 time units (invariant `clk<=1`). Messages sent by Source node are incrementally timestamped, from `1` to `MSGS`, which is the last message sent. After sending `MSGS` messages, the Source node stops transmitting.

Abstract model of a generic relay node.

The relay node is a template parametric with respect to the node identifier: `id`.



Abstract model of a generic relay node.

The node has only one location, 5 edges with different input actions ($chnl[0]? \cdots chnl[4]?$) and one edge with its own output action ($chnl[id]!$).

The output action `chnl[id]!bmessages[id]` represents the action executed by node `id` to broadcast the message `bmessages[id]`.

The input action `chnl[i]?` is the action executed by the node `id` for receiving a message from the node `i`.

Abstract model of a generic relay node.

- The output action `chnl[id]!` is enabled only if the global variable `bmessages[id]` contains at least one message to send, i.e. if `n > 0`.

  - `n` represents the number of messages to be forwarded

  - `buffer[0]` represents the messages stored in the head of the local buffer of the node `id`, which is the FIFO buffer that contains all the messages to be forwarded.

  - When a message is sent, it is stored into the global array `bmessages`, i.e. `bmessages[id] = buffer[0]`, then the function `update()` is executed for updating node's local data structures.

Auxiliary function.

```
void update() {
  n-;
 for( i :  size_t )
    buffer[i-1]=buffer[i];
  if ( m < DIM-1 ) {
   logger[m] = bmessages[id];
   m++;
  }
 }
```

After broadcasting a message, the function `update()` executes a backward one-position shift of `buffer`, and update the log recording the timestamp of the forwarded message.

Abstract model of a generic relay node.

- The input action `chnl[i]?` is always enabled. However, messages broadcasted by node `i` are received by node `id` only if nodes `i` and `id` are neighbors. The function `receive(id, i)` implements the receiving of messages from neighbors.

Auxiliary functions.

```
void receive(int j) {
  if (neighbor(id,j) && (bmessages[j]>TS) && (n<DIM-1)) {
     buffer[n] = bmessages[j];
     TS = bmessages[j];
     n++;
     }
  }
```

The test `bmessages[j] > TS` in the function `receive` evaluates `false` when the node `id` receives an old (already received) message.
In such cases, the received message is not stored into `buffer` and is dropped.

UPPAAL network

```
sourcenode := source();
node1 := relay(1);
node2 := relay(2);
node3 := relay(3);
node4 := relay(4);
system sourcenode, node1, node2, node3, node4;
```

Where `source()` and `relay(id)` represent the Source node and the Relay nodes, respectively.

Formalisation of the property

Recall that `checker_t` $\in$ `[0, MSGS-1]`, and `TS` $\in$ `[1, MSGS]`.

```
A<>( forall ( i:checker_t ) node1.logger[i] == i + 1 )
A<>( forall ( i:checker_t ) node2.logger[i] == i + 1 )
A<>( forall ( i:checker_t ) node3.logger[i] == i + 1 )
A<>( forall ( i:checker_t ) node4.logger[i] == i + 1 )
```

For each node, the buffer `logger` stores all the messages forwarded by it. Referring to the formulas above, `logger[i]` represents the `(i+1)`-th message that was forwarded by a certain node.

The *Property P* is proved to be true if, for each relay node, for each `i` such as `i` $\in$ `[0, MSGS-1)`, `logger[i]` stores the timestamp `i+1`. In this case, all nodes forwarded only once all the messages they received.

Modeling attacks

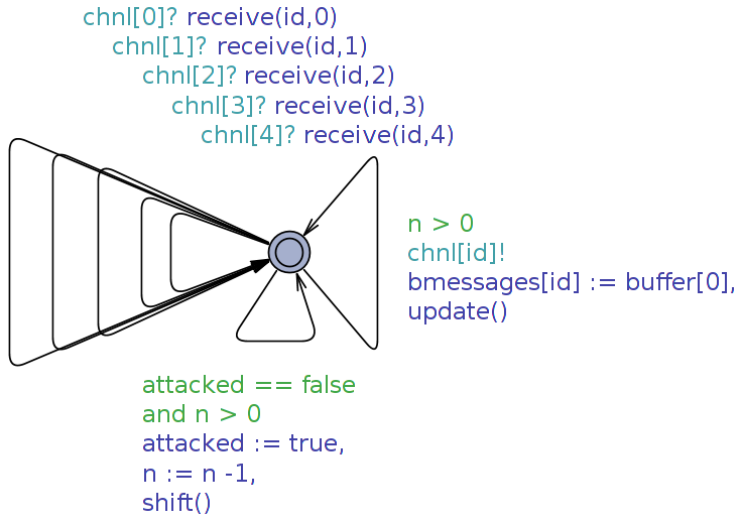
We consider two attacks in both of which a node is compromised by an adversary.

Drop attack
In the first attack, at a random time, the compromised node drops exactly one packet that, instead, should have been sent to its neighbors.

Tamper attack
In the second attack, the compromised node sends exactly one fake packet to its neighbors. Such a packet contains a fake timestamp, which is ahead in time compared to the current time. The reception of the fake packet causes, on the recipient, the discarding of all the genuine packets that carry a timestamp older than the fake one.

Both the attacks are modeled by adding exactly one transition to the model of the relay node. Both attacks occur at random time and execute only once.

Drop attack



`attacked` is a global flag initialised to `false` and set to `true` after the attack occurrence. Function `shift()` drops the message from the buffer

Drop attack

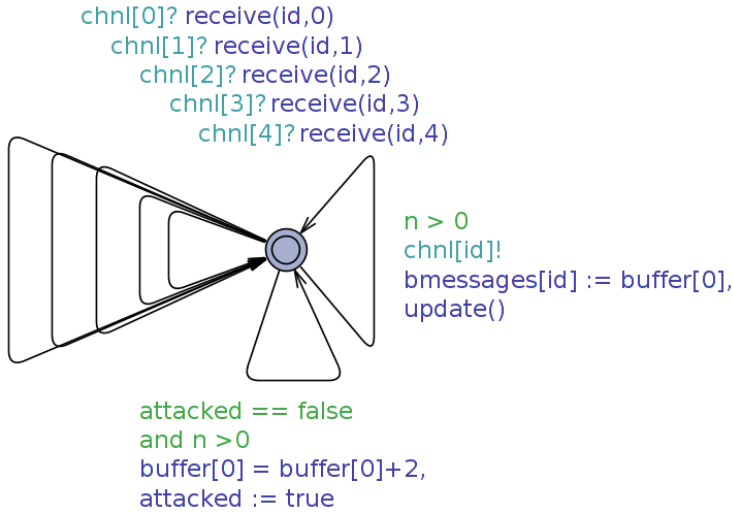
Simulations done via UPPAAL show the results that follow.

- when the adversary compromises Node 1, the *Property P* is still satisfied, since Node 3 receives a copy of the dropped packet from Node 2, thanks to link redundancy;

- when the adversary compromises Node 2, the *Property P* is not satisfied anymore because Node 4 does not receive a copy of the dropped packet since there is no redundancy on links connecting Node 4 with other network nodes.

Since any node can be attacked, the model checker returns FALSE.

Tampering attack



The timestamp of the message that is being to be broadcasted is advanced by 2.

Tampering attack

Simulations done via UPPAAL show the following results.

- When the adversary compromises Node 1, the *Property P* is not satisfied if Node 3 receives the fake packet from Node 1 before the genuine packets carrying timestamps older than the fake one from Node 2. The *Property* is satisfied if Node 3 receives from Node 2 all the genuine packets carrying timestamps older than the fake one before the fake packet from Node 1.

- When the adversary compromises Node 2, the *Property P* is not satisfied anymore because Node 4, after receiving the fake packet, discards all the subsequent genuine packets received from Node 4 that carry timestamps older than the fake one.

Since any node can be attacked, the model checker returns FALSE.

Flexibility of the modelling framework

An interesting property would be to prove whether a given topology is resilient to a given attack or not.

Moreover, the power consumption of a node caused by the reception of fake packets could be an interesting property to be analysed.

A packet injection attack with a given frequency could be modelled. Even if the packet injection attack does not interfere with the flooding protocol at application level, it results in draining the battery of the target node, thus reducing the network life-time.

Flexibility of the modelling framework

Timed automata have been used to formally analyse many security protocols.

UPPALL SMC is a new model ckecker, in which the clocks of an automaton may evolve with various rates. The network behaviour may depend on stochastic rates. For the analysis, statistical model checking is suggested, an alternative approach to an exhaustive exploration of the state-space model.

Model checking Cryptographic protocols

Cryptographic protocols work in an hostile environment

- are difficult to design, and are subject to subtle flaws, even when the cryptographic primitives are secure

In order to ensure that the protocol always achieve the goal, it is designed under the assumption that the network is completely controlled by an adversary (intruder)

The intruder can:
- intercept / redirect /alter data
- access to any operation of legitimate agents
- control one or more legitimate agents and access their keys

Set of slides based on the book chapter: D. Basin, C. Cremers, C. Meadows. Model checking Security protocols. E.M.Clark et al. (eds.), Handbook of Model Checking, Springer Int. Publishing, 2018.

Needham-Schroeder public key protocol

The goal of the protocol is to allow two agents, A and B, to authenticate each other.

We use asymmetric encryption: given an agent j,
- $pk(j)$ is the public key of j, and
- $sk(j)$ is the private key of j.

Any agent can send a message to agent j using the public key of j for encryption
Only the agent j can decrypt the message using its private key.

At the end of the protocol A and B agree on a the pair of secrets
- N_A, a nounce generated by agent A (a random number), and
- N_B, a nounce generated by agent B (random number).

Needham-Schroeder public key protocol

Steps of the protocol
$\{| \cdots |\}^a$ stands for asymmetric encryption

*Step*1. *A generates* N_A
$A \to B : \{|\ A, N_A\ |\}^a_{pk(B)}$

*Step*2. *B decrypts the message*
B generates N_B
$B \to A : \{|\ N_A, N_B\ |\}^a_{pk(A)}$

*Step*3. *A decrypts the message*
$A \to B : \{|\ N_B\ |\}^a_{pk(B)}$

Step 2. *B* assumes that the message has been sent by *A* recently
Step 3. *A* knows that the received message has been sent recently (it contains N_A)
A confirms the assumption of *B*, sending a message with N_B

Lowe's man-in-the-middle attack

Agent a wants to communicate with agent i. Agent i is malicious and uses nonce of a to impersonate a and to communicate with b

$Step1.\quad a \to i : \{|\, a, N_a \,|\}^a_{pk(i)}$

$Step2.\quad i \to b : \{|\, a, N_a \,|\}^a_{pk(b)}$

$Step3.\quad b \to a : \{|\, N_a, N_b \,|\}^a_{pk(A)}$

$Step4.\quad a \to i : \{|\, N_b \,|\}^a_{pk(i)}$

$Step5.\quad i \to b : \{|\, N_b \,|\}^a_{pk(b)} \qquad (*)$

$(*)$: the intruder re-encrypts N_b under b public key and sends it to b.
b concludes that he shares N_a and N_b with a and a only.

Lowe's man-in-the-middle attack

- non-intuitive attack
- security relies on the assumption that agents do not reveal secrets (violated by i, which sends the nounce of a to b)
- the attack does not depends on flaws in the cryptographic primitives
- the attack requires a very simple adversary model

Formal model of the adversary (Dolev-Yao approach)

check correctness of protocols with respect to more powerful adversaries

Actions available to the adversary:
- read, redirect and delete messages
- impersonate any agent
-create new messages applying functions available to previously seen data

This model allows to capture many non-intuitive security flaws

Model checking can be used for the analysis of the protocol with this model of the adversary

Formal model of the adversary (Dolev-Yao approach)

In the Dolev-Yao approach,

- a protocol is modelled as a machine consisting of an arbitrary number of honest agents executing the protocol
- all messages sent are intercepted by the adversary
- all messages received are sent by the advesary
- Any message processing done by the adversary is done using an arbitrary combination of a finite set of operations
- messages are terms rather than strings
- cryptography is perfect
- the only way an adversary has to decript a message is to have the decryption key.

Formal model of the adversary (Dolev-Yao approach)

Dolev and Yao proposed polynomial algorithms for proving security of ping-pong protocols:
- protocols that involve two participants, the sender S and the receiver R.
- in each step, the participant applies a sequence of operators to the last message received, and sends it back.

Sometimes protocols are vulnerable only to a number of interleaved sessions.

The secrecy problem is
- undecidable for an unbounded number of sessions and nounces
- NP-complete if the number of sessions is bounded.

Bounded-session model checkers (the user the number of sessions) allows to analyse realistic protocols.

Formal model of cryptopgraphic protocols

Tools are based on formal models of cryptographic protocols and actions available to the adversary.

A basic symbolic model

- messages are represented by terms in a term algebra
- protocols are described by a set of roles
  each role is described by a sequence of events taken by the agents executing the role
- agents may play in multiple roles

In the following, a basic model is presented.

Terms

$$BasicTerm ::= Agent \mid Role \mid Fresh \mid Var \mid AdvConst \mid Fresh\sharp TID \mid Var\sharp TID$$

AdvConst: adversary generated constants

$tid \in TID$ is the identifier of a thread.
The notation $t\sharp tid$ binds variables and freshly generated terms to threads: the term t is local to the protocol role instance denoted by thread identifier tid.

$$Terms ::= BasicTerm \mid (Term, Term) \mid pk(Term) \mid sk(Term) \mid k(Term, Term) \mid \{|Term|\}^{a}_{Term} \mid \{|Term|\}^{s}_{Term} \mid Func(Term^{\star})$$

$pk(X)$ is the public key of X
$sk(X)$ is the secret key of X
$k(X, Y)$ is the symmetric key belonging to X and Y.
Func is used for defining other cryptograhic functions.

Let t^{-1} be the inverse key of t. We have:
$pk(X)^{-1} = sk(X)$, $sk(X)^{-1} = pk(X)$ $\forall X \in Agents$, $t^{-1} = t$ for all other terms t.

Power of the adversary

Let $\vdash$ a binary relation where $M \vdash t$ means that t can be inferred from the set of terms in M

$\vdash$ is the smallest relation statisfying:.

$t \in M \Rightarrow M \vdash t$
$M \vdash t_1 \wedge M \vdash t_2 \Leftrightarrow M \vdash (t_1, t_2)$
$M \vdash t_1 \wedge M \vdash t_2 \Rightarrow M \vdash \{|t_1|\}^a_{t_2}$
$M \vdash t_1 \wedge M \vdash t_2 \Rightarrow M \vdash \{|t_1|\}^s_{t_2}$
$M \vdash \{|t_1|\}^s_{t_2} \wedge M \vdash t_2 \Rightarrow M \vdash t_1$
$M \vdash \{|t_1|\}^a_{t_2} \wedge M \vdash (t_2)^{-1} \Rightarrow M \vdash t_1$
$\wedge_{0 \leq i \leq n} M \vdash t_i \Rightarrow M \vdash f(t_0, \cdots, t_n)$

Subterms t of a term t', written $t \sqsubseteq t'$, are the syntactic subterms of t'
for example, $t_1 \sqsubseteq \{|t_1|\}^s_{t_2}$ and $t_2 \sqsubseteq \{|t_1|\}^s_{t_2}$
$FV(t)$ are the free variables of t:
$FV(t) = \{t' \mid t' \sqsubseteq t \wedge t' \in Var \cup \{v \sharp tid \mid v \in Var \wedge tid \in TID\}\}$

Events

The events are (concurrent execution of processes): starting a thread, sending a message, receiving a message.

$$Event ::= create(Role, \mathcal{L}ub) \mid send(Term) \mid recv(Term)$$

- $\mathcal{L}ub$ is a set of substitutions of terms for variables: $[t_0, \cdots, t_n/x_0, \cdots, x_n] \in \mathcal{L}ub$

- *send* and *receive* do not include explicit sender or receiver (sender and receivers can be subterms in the message sent/received)

- let σ be a substitution: $\sigma(send(m)) = send(\sigma(m))$

- The adversary receives all messages sent, independently of the intended recipient.

Protocols

- a protocol is a mappping from role names to event sequence

$Protocol : Role \nrightarrow Events^{\star}$

Consider two roles, Initiator and Recipient.
$\{Init, Recp\} \subseteq Role$, $key \in Fresh$, $x \in Var$

Example of protocol

$P(Init) = \langle send(Init, Recp, \{|\{|Recp, key|\}^{a}_{sk(Init)}|\}^{a}_{pk(Recp)})\rangle$

$P(Recp) = \langle recv(Init, Recp, \{|\{|Recp, key|\}^{a}_{sk(Init)}|\}^{a}_{pk(Recp)})\rangle$

Protocols

- protocols are executed by agents who executes roles
- role names are assigned agent names

Variables, fresh terms and roles of each thread are assigned unique names by *localize* : $TID \rightarrow \mathcal{L}ub$

$localize(tid) = \bigcup_{cv \in Var \cup Fresh}[cv\sharp tid/cv]$

- *thread* gives the sequence of agent events that may occur in a thread. The domain of substitution ($\mathcal{L}ub$) is extended to roles

thread : $(Events^{\star} \times TID \times \mathcal{L}ub) \rightarrow Events^{\star}$

Let l be a sequence of events, $tid \in TID$, and let σ be a substitution

$thread(l, tid, \sigma) = \sigma(localize(tid)(l))$

Threads

Consider the protocol

$P(Init) = \langle send(Init, Recp, \{|\{|Recp, key|\}^a_{sk(Init)}|\}^a_{pk(Recp)})\rangle$

$P(Recp) = \langle recv(Init, Recp, \{|\{|Recp, x|\}^a_{sk(Init)}|\}^a_{pk(Recp)})\rangle$

Let $t1 \in TID$ and $\{A, B\} \subseteq Agent$

Assume thread t_1 performs the Init role. We have:

$localize(t_1)(key) = key\sharp t_1$, and

$thread(P(Init), t_1, [A, B/Init, Recp]) = \langle send(A, B, \{|\{|B, key\sharp t_1|\}^a_{sk(A)}|\}^a_{pk(B)})\rangle$

with $[A, B/Init, Recp] \in \mathcal{L}ub$ the assignment to role names of agents

Adversary knowledge

Initial knowledge

- The adversary knows all agents and the public key of the agents.
- The adversary can generate constants, using functions.
  The set *AdvConst* is the set of fresh values the adversary can generate.
- If the adversary has compromised some agents, the adversary additionally knows the private key of such agents.

For any agent a,
$Keys(a) = \{sk(a)\} \cup \bigcup_{b \in Agent} \{k(a,b), k(b,a)\}$.

We divide agents into Honest agents and Compromised agents.

Adversary initial knowledge IK_0:

$$IK_0 = Agent \cup AdvConst \cup \bigcup_{a \in agent} \{pk(a)\} \cup \bigcup_{b \in Compromised} \{sk(b)\}$$

Model checking Cryptograhic protocols

Model checking cryptograhic protocols started with Lowe's use of FDR model checker to analyse the Needham-Schroeder public-key protocol, which demonstrate a problem unnoticed for many years.

Book chapter: D. Basin, C. Cremers, C. Meadows. Model checking Security protocols. E.M.Clark et al. (eds.), Handbook of Model Checking, Springer Int. Publishing, 2018.

Execution model

The execution model is a transition system, with an associated notion of traces.

A trace is a possible execution history:
$Trace = (TID \times Event)^\star$

A state is a tuple (tr, IK, th) where
- tr is a trace
- IK is the adversary knowledge
- th is a partial function mapping thread identifiers of executing or completed threads to event traces. This is useful to define the functionality of threads.

$State = Trace \times \mathcal{P}(Term) \times (TID \nrightarrow Event^\star)$

Initial system state: $s_{init} = (\langle\rangle, IK_0, \emptyset)$

Semantics

The semantics of a protocol P is a transition system generated by rules.
Each rule describe the execution of an event: *create*, *send* and *receive*.

Rule for $create_P$

$$\frac{R \in dom(P) \quad \sigma \in dom(P) \to Agent \quad tid \notin dom(th)}{(tr, IK, th) \to (tr \cdot \langle (tid, create(R, \sigma)), IK, th[tid \vdash thread(P(R), tid, \sigma)])}$$

Create a new instance of a protocol role R (thread)

Premises:
- σ associates the role names ($dom(P)$) with agents ($Agent$)
- tid is a fresh thread identifier ($tid \notin dom(th)$)

Consequences:
- the successor trace is extended, reflecting that the thread tid executed the create event and
- the thread mapping is extended with the thread assigned to tid

Semantics

Rule for *send*

$$\frac{th(tid) = \langle send(m) \rangle \cdot l}{(tr, IK, th) \rightarrow (tr \cdot \langle (tid, send(m)) \rangle, IK \cup \{m\}, th[tid \mapsto l])}$$

send sends a message m to the *network*. The message is added to IK.

Premises:
- next event of thread *tid* is the send of m

Consequences:
- the successor trace is extended, reflecting the send event
- the adversary knowledge is updated with m
- thread for *tid* is updated

Note that IK includes all messages sent by agents.

Semantics

Rule for *recv*

$$\frac{th(tid) = \langle recv(pt) \rangle \cdot l \quad IK \vdash \sigma(pt) \quad dom(\sigma) = FV(pt)}{(tr, IK, th) \rightarrow (tr \cdot \langle (tid, recv(\sigma(pt))) \rangle, IK, th[tid \mapsto \sigma(l)])}$$

Models an agent running a thread that receives a message from the network

Premises:
- the message must match the message pattern *pt* under a substitution σ (*pt* may contain free variables)
- $\sigma(pt)$ must be inferred from knowledge *IK*.

Consequences:
- recipients accept all messages that match the message pattern *pt*, and block on any other messages.
- the resulting σ is applied to the remaining protocol steps *l* (in the rule: $th[tid \mapsto \sigma(l)]$)

Note that recipients accept all messages that match the message pattern *pt*, any other message is blocked.

Transition relation

Definition (Transition relation)

Let P be a protocol. We define the transition relation $\rightarrow_P$ as follows:

given s and s', $s \rightarrow_P s'$ iff there exists a rule with the premises $Q_1(s), \cdots, Q_n(s)$ and the conclusions $s \rightarrow s'$ such that all the premises hold.

Definition (Reachable states)

Given a set of states T:

$$Post_P(T) = \{s' \in States \mid \exists s \in T : s \rightarrow s'\}$$

$Reachable(P) = \bigcup_{n=0,\cdots,\infty} Post_P^n(\{s_{init}\})$.

Needham-Schroeder Protocol

$A, B \in Agent$

$$
\begin{aligned}
P = \quad & Init \rightarrow Recp : \{\!|\ Init, N_{Init}\ |\!\}^{a}_{pk(Recp)} \\
& Recp \rightarrow Init : \{\!|\ N_{Init}, N_{Recp}\ |\!\}^{a}_{pk(Init)} \\
& Init \rightarrow Recp : \{\!|\ N_{Recp}\ |\!\}^{a}_{pk(Recp)}
\end{aligned}
$$

$thread(P(Init), t_1, [A, B/Init, Recp]) =$
$snd(A, B, \{\!|\ A, key_{\sharp t_1}\ |\!\}^{a}_{pk(B)})$
$rcv(A, B, \{\!|\ key_{\sharp t_1}, x_{\sharp t_1}\ |\!\}^{a}_{pk(A)})$
$snd(A, B, \{\!|\ x_{\sharp t_1}\ |\!\}^{a}_{pk(B)})$

$thread(P(Recp), t_2, [A, B/Init, Recp]) =$
$rcv(A, B, \{\!|\ A, x_{\sharp t_2})^{a}_{pk(B)})$
$snd(A, B, \{\!|\ x_{\sharp t_2}, key_{\sharp t_2}\ |\!\}^{a}_{pk(A)})$
$rcv(A, B, \{\!|\ key_{\sharp t_2}\ |\!\}^{a}_{pk(B)})$

Needham-Schroeder Protocol: an execution

$s = (tr, IK, th)$
$s_{init} = (\langle\rangle, IK_0, \emptyset)$, with $IK_0 = \emptyset$

$$(\langle\rangle, IK_0, \emptyset)$$
$$\downarrow_{create(t1)}$$

$$(\langle(t1, crt(A, [A, B/Init, Recp]))\rangle, IK_0, (t1 : snd; rcv; snd))$$
$$\downarrow_{create(t2)}$$

$$(\langle ...(t2, crt(B, [A, B/Init, Recp]))\rangle, IK_0, \{t1 : snd; rcv; snd / t2 : rcv; snd; rcv\}$$
$$\downarrow_{snd(t1)}$$

$$(\langle ...(t1, snd(...))\rangle, \{\{|\ A, key_{\sharp t_1}\ |\}^a_{pk(B)}\}, \{t1 : rcv; snd / t2 : rcv; snd; rcv\})$$
$$\downarrow_{rcv(t2)}$$

$$(\langle ...(t2, rcv(...))\rangle, \{\{|\ A, key_{\sharp t_1}\ |\}^a_{pk(B)}\}, \{t1 : rcv; snd / t2 : snd : rcv[x_{\sharp t_2} = key_{\sharp t_1}]\})$$
$$\downarrow_{snd(t2)}$$

$$(\langle ...(t2, snd(...))\rangle, \{\{|\ A, key_{\sharp t_1}\ |\}^a_{pk(B)}, \{|\ key_{\sharp t_1}, key_{\sharp t_2}\ |\}^a_{pk(A)}\}, \{t1 : rcv; snd / t2 : rcv[x_{\sharp t_2} = key_{\sharp t_1}]\})$$

Needham-Schroeder Protocol

$$(\langle ...(t2, snd(...))\rangle, \{\{|\ A, key_{\sharp t_1}\ |\}^a_{pk(B)}, \{|\ key_{\sharp t_1}, key_{\sharp t_2}\ |\}^a_{pk(A)}\}, \{t1 : rcv; snd/t2 : rcv[x_{\sharp t_2} = key_{\sharp t_1}]\})$$
$$\downarrow_{rcv(t1)}$$
$$(\langle ...\rangle, \{\{|\ A, key_{\sharp t_1}\ |\}^a_{pk(B)}, \{|\ key_{\sharp t_1}, key_{\sharp t_2}\ |\}^a_{pk(A)}\}, \{t1 : snd[x_{\sharp t_1} = key_{\sharp t_2}]/t2 : rcv[x_{\sharp t_2} = key_{\sharp t_1}]\})$$
$$\downarrow_{snd(t1)}$$
$$(\langle ...\rangle, \{\{|\ A, key_{\sharp t_1}\ |\}^a_{pk(B)}, \{|\ key_{\sharp t_1}, key_{\sharp t_2}\ |\}^a_{pk(A)}, \{|\ key_{\sharp t_2}\ |\}^a_{pk(B)}\}, \{t1 : \langle\rangle/t2 : rcv - [x_{\sharp t_2} = key_{\sharp t_1}]\})$$
$$\downarrow_{rcv(t2)}$$
$$(\langle ...\rangle, \{\{|\ A, key_{\sharp t_1}\ |\}^a_{pk(B)}, \{|\ key_{\sharp t_1}, key_{\sharp t_2}\ |\}^a_{pk(A)}, \{|\ key_{\sharp t_2}\ |\}^a_{pk(B)}\}, \{t1 : \langle\rangle/t2 : \langle\rangle\})$$

Needham-Schroeder Protocol

$$thread(P(Init), t_1, [A, B/Init, Recp]) =$$
$$snd(A, B, \{|\ A, key_{\sharp t_1}\ |\}^a_{pk(B)})$$
$$rcv(A, B, \{|\ key_{\sharp t_1}, x_{\sharp t_1}\ |\}^a_{pk(A)})$$
$$snd(A, B, \{|\ x_{\sharp t_1}\ |\}^a_{pk(B)})$$

$$thread(P(Recp), t_2, [A, B/Init, Recp]) =$$
$$rcv(A, B, \{|\ A, x_{\sharp t_2})^a_{pk(B)})$$
$$snd(A, B, \{|\ x_{\sharp t_2}, key_{\sharp t_2}\ |\}^a_{pk(A)})$$
$$rcv(A, B, \{|\ key_{\sharp t_2}\ |\}^a_{pk(B)})$$

Lowe's man-in-the-middle attack

Reconsider the Lowe's attack to Needham-Schroeder public key protocol.

*Step*1. $a \to i : \{|\ a, N_a\ |\}^a_{pk(i)}$

*Step*2. $i \to b : \{|\ a, N_a\ |\}^a_{pk(b)}$

*Step*3. $b \to a : \{|\ N_a, N_b\ |\}^a_{pk(A)}$

*Step*4. $a \to i : \{|\ N_b\ |\}^a_{pk(i)}$

*Step*5. $i \to b : \{|\ N_b\ |\}^a_{pk(b)} \qquad (*)$

$(*)$: the intruder re-encrypts N_b under b public key and sends it to b.
b concludes that he shares N_a and N_b with a and a only.

Properties

Agents= $\{A, B, i\}$

We divide agents into *Honest* agents and *Compromised* agents.

$Honest = \{A, B\}$ $Compromised = \{i\}$

$IK_0 = \{pk(A), pk(B), pk(I), sk(i)\}$

$thread(P(Init), t1, [A, i/Init, Recp])$

$thread(P(Recp), t2, [A, B/Init, Recp])$

the compromised agent may have the role Init or Recp with agent A or agent B and use the *IK* to generate terms and to infer new terms.

Properties

Definition (Secrecy)

Let $t \in \mathit{Fresh}$. We say that a state $s = (tr, IK, th)$ satisfies secrecy of t if and only if

$\forall tid$, if tid is a *Honest* thread, then $\neg(IK \vdash (t \sharp tid))$

A protocol P ensures secrecy of t if an only if all reachable states of P satisfy secrecy of t.

The Needham-Schroeder public key protocol does not ensure the secrecy of the nouces.

Model checking algorithms

The simplicity of the Dolev-Yao adversary has made it popular.

For many real-world scenarios, an adversary who has complete control of the network may can unrealistic.

Main issues in developing model checkers for cryptographic protocols:
- forward and backward search
- state representations
- bounding the state space

Specialized techniques have been developed.

Some tools: NRL Protocol Analyzer (NPA), Maude-NPA, AVISPA, ProVerif tool, ...