[Pervasive Computing exercises] Report

Federico Mazzini
federico.mazzini3@studio.unibo.it

Marzo 2022

1 Lab 02

1.1 The verifier task

Code and do some analysis on the Readers & Writers Petri Net. Add a test to check that in no path long at most 100 states mutual exclusion fails (no more than 1 writer, and no readers and writers together). Can you extract a small API for representing safety properties?

Below is the graphical representation for the petri net concerning Readers and Writers.

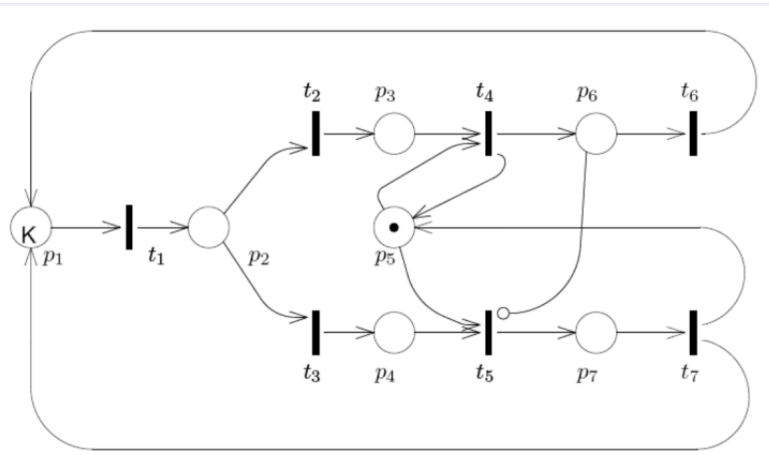

Figure 1: Readers and Writers Petri Net

At first, I create *PNReadersWrites* source to model states and transitions using the provided Scala DSL. Then I created an API that allows to check *safety* and *liveness* property. Finally in *PNReadersWritersTest* source, I tried to verify that in no path mutual exclusion fails, that is two or more writers at the same time, or a writer at the same time with one or more readers.

1.1.1 Petri Net in Scala

I tried to give semantics to the various places with more declarative names.

```
object place extends Enumeration {
    val START, CHOICE, ASKREAD, ASKWRITE, PERMIT, READ, WRITE = Value
  }

def readersWritersSystem() = toSystem(PetriNet[Place](
    MSet(START) ~~> MSet(CHOICE),
    MSet(CHOICE) ~~> MSet(ASKREAD),
    MSet(CHOICE) ~~> MSet(ASKWRITE),
    MSet(ASKREAD, PERMIT) ~~> MSet(PERMIT, READ),
```

```
    MSet(ASKWRITE, PERMIT) ~~> MSet(WRITE) ^^^ MSet(READ),
    MSet(READ) ~~> MSet(START),
    MSet(WRITE) ~~> MSet(PERMIT, START)
  ))
```

1.1.2 API

The API has been modeled by exploiting Scala implicit, pimping the return type when requesting the paths for a certain system. The paths related to different execution scenarios are collected in a stream. This data structure is pimped with two methods *checkSafety* and *checkLiveness* which, starting from the conditions to be verified, scan the states of each path and proceed to the verifications.

```
object SystemVerifies{
  implicit class PathsExtension[S](paths: Stream[List[S]]){
    def checkSafety(condition: S => Boolean) =
      verifyAlways(paths)(condition)

    def checkLiveness(condition: S => Boolean) =
      verifyCertain(paths)(condition)
  }

  private def verifyAlways[S](paths: Stream[List[S]])(condition: S => Boolean) =
    paths.foreach(path =>
      path.foreach(state => assert(condition(state))))

  private def verifyCertain[S](paths: Stream[List[S]])(condition: S => Boolean) =
    paths.foreach(path =>
      assert(path.exists(state => condition(state))))
}
```

1.1.3 Test safety properties

Having defined the API to check *safety* and *liveness* properties, a possible use is the following, in which I try to understand if in any possible path the mutual exclusion is never violated.

```
"PN for readers and writers" should
"avoid more than 1 writer at the time and no reader" in {
   //without API
   PNRW.completePathsUpToDepth(MSet(START, START, START, PERMIT), 100)
     .foreach(path => path
       .foreach(state =>
                        //only one writer concurrently
                        ! (state matches MSet(WRITE, WRITE)) &&
```

```
                        //no reader allowed when writer writes
                        ! (state matches MSet(WRITE, READ))
  ))

   //with API
   PNRW.completePathsUpToDepth(MSet(START, START, START, PERMIT),100)
     .checkSafety {
       state => !(state matches MSet(WRITE, WRITE)) &&
                !(state matches MSet(WRITE, READ))
   }
 }
```

Unfortunately, running checks in a path of up to 100 states takes a lot of computation time. So I reduced the path limits to 15.

The property holds, mutual exclusion doesn't fail.

1.1.4 Test liveness property

During the API development it was decided to also define a function for verifying *liveness* properties. This property states that something good will eventually occur in the system. In this usage example I try to verify if the system, in all execution scenarios (not in every state), will be able to perform a READ.

```
"PN for readers and writers" should
"permit to read at some point" in {
    PNRW.paths(MSet(ASKWRITE,ASKREAD,PERMIT),  15) checkLiveness  {
      state => state matches MSet(READ)
    }
  }
```

This property is not verified because it exists a scenario in which the system loops over the token in ASKWRITE. A process, or a group of processes, continues to READ and maintain the permit token.

From this check we get the fact that there is no *fairness* within the system.

1.2 The designer task

Code and do some analysis on a variation of the Readers & Writers Petri Net: it should be the minimal variation you can think of, such that if a process says it wants to read, it eventually (surely) does so. How would you show evidence that your design is right?

I initially tried to figure out how a reader might want to read but can't.

A reader cannot read if there are no tokens in the place P_5/PERMIT of the previous page. The only way for this to happen is the fact that it has been consumed by a writing

process. Never being able to read means that a series of writers continuosly want to write during the simulation, always managing to consume the token in P_5/PERMIT.

I have therefore tried to avoid this situation by inserting an inhibitor arc from P_3/ASKREAD to T_0. In this way, when even just one process requests to read (arrives in P_3/ASKREAD), it will inhibit the transition T_0. No other process will be able to perform the T_0 transition as long as there are processes in P_3/ASKREAD, ensuring that any process it wants to read will be able to do so.

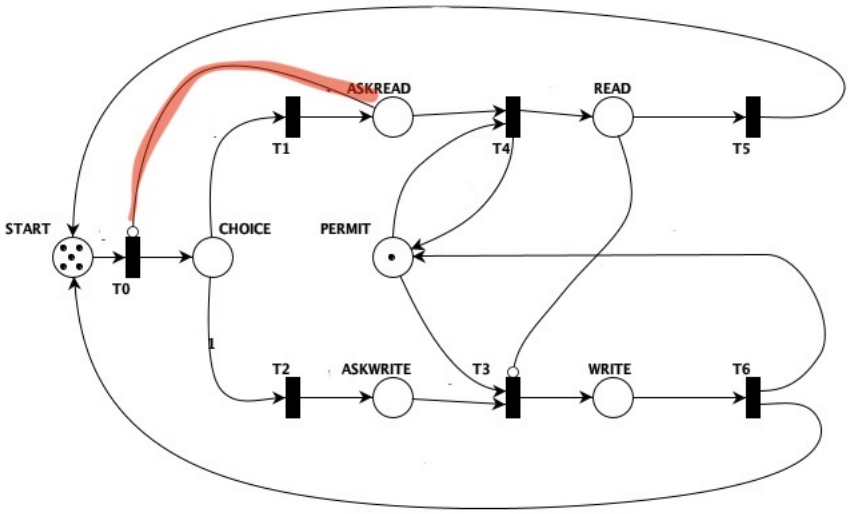

Figure 2: Readers and Writers variation

```
/** Variation of reader and writers problem such that if a process
says it wants to read, it eventually (surely) does so */
  def readersWritersSystemReadSurely() = toSystem(PetriNet[Place](
    MSet(START) ~~> MSet(CHOICE) ^^^ MSet(ASKREAD),
    MSet(CHOICE) ~~> MSet(ASKREAD),
    MSet(CHOICE) ~~> MSet(ASKWRITE),
    MSet(ASKREAD, PERMIT) ~~> MSet(PERMIT, READ),
    MSet(ASKWRITE, PERMIT) ~~> MSet(WRITE) ^^^ MSet(READ),
    MSet(READ) ~~> MSet(START),
    MSet(WRITE) ~~> MSet(PERMIT, START)
  ))
```

1.2.1 Verification

The check was done using the API defined in the verifier task. I defined a test to verify that if a process says it wants to read, eventually does so. This property is defined

liveness property.

```
"If a process says it wants to read" should
    "eventually (surely) does so" in {
    PNRWReadSurely.paths(MSet(START,ASKREAD,PERMIT),  15) checkLiveness  {
      state => state matches MSet(READ)
    }
  }
```

1.2.2 Another Readers and Writers variation

The idea is that a process that wants to write takes precedence over a process that wants to read. This idea is developed using an inhibitory arc from P_4/ASKWRITE to the transition T_4.

This arc prevents the firing of the T_4 transition when there is at least one token in P_4/ASKWRITE, so as long as there are processes asking to write, they will do so and take precedence over the processes they want to read.

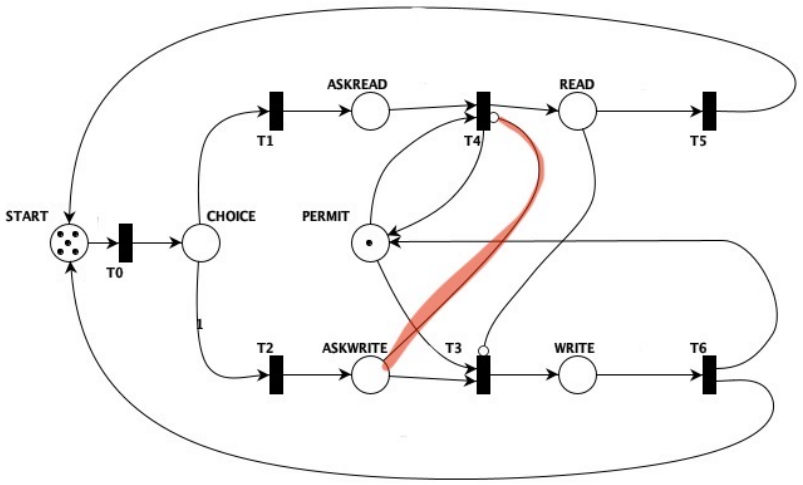

Figure 3: Readers and Writers variation

```
/** Variation of readers and writers problem in which
writer processes have priority over reader processes*/
  def readersWritersSystemWritePriority() = toSystem(PetriNet[Place](
    MSet(START) ~~> MSet(CHOICE),
    MSet(CHOICE) ~~> MSet(ASKREAD),
    MSet(CHOICE) ~~> MSet(ASKWRITE),
    MSet(ASKREAD, PERMIT) ~~> MSet(PERMIT, READ) ^^^ MSet(ASKWRITE),
```

```
    MSet(ASKWRITE, PERMIT) ~~> MSet(WRITE) ^^^ MSet(READ),
    MSet(READ) ~~> MSet(START),
    MSet(WRITE) ~~> MSet(PERMIT, START)
  ))
```

To verify the property, I define a test which checks that in each i state of the system in which there are two processes that want to read and write, there isn't $i + 1$ state in which there is a reading (because writing must always firing first).

```
"PN for readers and writers with write priority" should
    "give priority to process that want write" in {
    PNRWWritePriority.paths(MSet(ASKREAD, ASKWRITE, PERMIT), 10).foreach {
      path => path
        .sliding(2, 1).map(l => (l(0), l(1)))
        .filter(state => state._1 matches MSet(ASKWRITE, ASKREAD))
        .foreach(state => assert(!(state._2 matches MSet(ASKWRITE, READ))))
    }

    PNRWWritePriority.paths(MSet(ASKREAD, ASKWRITE, PERMIT), 10) checkLiveness {
      state => state matches MSet(WRITE)
    }
  }
```

In the second part of the test I check that in all execution scenarios, a process that wants to write eventually does so.

1.3 The simulator task

Take the communication channel CTMC example in StochasticChannelSimulation.

- Compute the average time at which communication is done—across n runs.
- Compute the relative amount of time (0% to 100%) that the system is in fail state until communication is done—across n runs.
- Extract an API for nicely performing similar checks.

1.3.1 CTMC

(Continuous Time Markov Chain) is used as an extension for a transition system in a continuous environment. Time is no longer modeled as a sequence of discretes steps.

Starting from the *rate* in each transition, a **CDF** (Cumulative Distribution Function) is used to get the probability that an event happens before a certain time t. The higher the markovian rate, the higher the probability that the transition will be made sooner. Hence in this process transitions are probabilistic (after normalization) and timed.

Starting from figure 4, the properties that characterize a transition system with a CTMC process are the following:

- **Memoryless** (Markovian Property) The future is independent of the past. A subsequent transition is not affected by previous ones.
- Duration in S_0 is Markovian with rate $\sum r_i$
- The probability of transiting from S_0 to S_i is $r_i / \sum r_i$

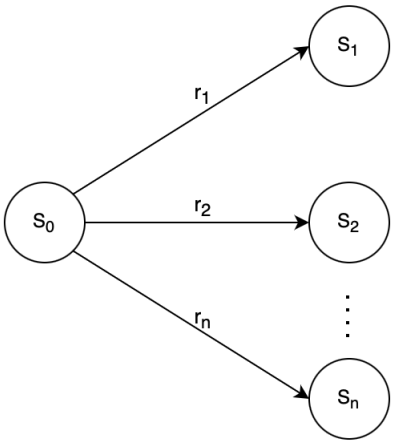

Figure 4: Example for explaining Markovian Rates

1.3.2 Communication Channel

The system is defined by a series of transitions, each defined by a starting state, an arrival state and a *markovian rate*.

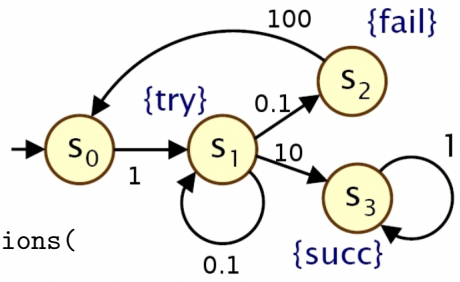

Figure 5: Communication Channel (rates are wrong)

```
object state extends Enumeration {
    val IDLE,SEND,DONE,FAIL = Value
  }

def stocChannel: CTMC[State] = CTMC.ofTransitions(
  (IDLE,1.0,SEND),
  (SEND,100000.0,SEND),
  (SEND,200000.0,DONE),
  (SEND,100000.0,FAIL),
  (FAIL,100000.0,IDLE),
  (DONE,1.0,DONE)
)
```

1.3.3 API

I initially developed a small API to declare n run quickly. A simulation takes as input the number of runs, the maximum length for them and a function to define them. Hence, doing a simulation means creating a stream of different runs, on which to perform checks.

It was also defined a *reduce* function in order to manipulate the stream before checking the properties.

```
object Simulation {
  def fromTrace[A](n: Int,
                   maxTraceLength: Int,
                   trace: () => Iterable[(Double, A)]): Simulation[A] =
    Simulation(n, maxTraceLength, trace)

  case class Simulation[A](n: Int,
                           maxTraceLength: Int,
                           trace: () => Iterable[(Double, A)]) {

    val traces: Iterable[Iterable[(Double, A)]] =
        (1 to n).map(_ => trace().take(maxTraceLength).toList)

    def reduce[B](f: (Iterable[(Double, A)] => B)): Iterable[B] = traces.map(f(_))
  }
}
```

In this example, I defined a simulation composed by *10000 runs*, each long *100 steps*

```
val channelAnalysis = CTMCSimulation(channel)

val simulation =  Simulation.fromTrace(10000,
                                       100,
                                       () => channelAnalysis.
                                              newSimulationTrace(IDLE, new Random()))
```

The resulting simulation is a stream of traces. Each trace is called run and it's itself a stream of *(time, state)* tuples.

1.3.4 Average time communication done

To compute the average time at which the communication is done, it's necessary to find the first *done* state in each run, using the reduce function. Since each run is a stream of *tuples (time, state)*, knowing the first state *done* means knowing the time in which this happened.

```
val doneTimes = simulation.reduce(_.dropWhile(_._2 != DONE).head._1)

val avgTime = doneTimes.sum / doneTimes.size
val maxTime = doneTimes.max
val minTime = doneTimes.min
```

1.3.5 Amount of fail time until communication done

This is the relative amount of time that the system is in fail state until communication is done—across n runs.

```
val failTime = simulation.reduce{ run =>
    run.toList
      .takeWhile(_._2 != DONE)
      .sliding(2, 1)
      .collect { case (t1, FAIL) :: (t2, _) :: Nil => t2 - t1 }
      .sum
  }

val avgFailTime = failTime.sum / failTime.size
val failTimePercentage = (avgFailTime / avgTime * 100).round
```

Results for standard configuration (slide):

```
AVG DONE time: 1.502642259499749
MAX DONE time: 13.080744396820528
MIN DONE time: 7.936852284468996E-5
AVG FAIL time: 5.190826994624033E-6
% of FAIL time before DONE : 0%
```

The time passed in fail state is very low compared to the done time, this is due to the very high markovian rates except in the first transition, where most of the time is consumed. Below is another configuration with different markovian rates, which result in different average *fail* times.

```
def stocChannel2: CTMC[State] = CTMC.ofTransitions(
    (IDLE,1.0,SEND),
    (SEND,2.0,SEND),
    (SEND,4.0,DONE),
    (SEND,1.0,FAIL),
    (FAIL,1.0,IDLE),
    (DONE,1.0,DONE)
  )
```

```
AVG DONE time: 1.7280885496801852
MAX DONE time: 15.30154351618515
MIN DONE time: 0.009903227528155126
AVG FAIL time: 0.24813299695287527
% of FAIL time before DONE : 14%
```

1.4 The artist task

Create a variation/extension of PetriNet design, with **priorities**: each transition is given a numerical priority, and no transition can fire if one with higher priority can fire. Show an example that your pretty new "abstraction" works.

1.4.1 New PetriNet Model

The Petri Net object in the *PetriNet.scala* source has been converted to a *trait* and extended by two *object*. The first maintains the characteristics of the petri net model already present in the engine. The second extends it with the logic of priorities.

A new *type* **Transition** has been defined, its definition is left to the petri net specializations.

```
trait PetriNet{

  type Transition[P]
  type PetriNet[P] = Set[Transition[P]]

  def apply[P](transitions: Transition[P]*): PetriNet[P] =
    transitions.toSet

  def toPartialFunction[P](pn: PetriNet[P]): PartialFunction[MSet[P],Set[MSet[P]]]

  def toSystem[P](pn: PetriNet[P]): System[MSet[P]] =
    System.ofFunction( toPartialFunction(pn))
}
```

The implementation of the Priority Petri Net model is as follows

```
trait Priority extends PetriNet {
  override type Transition[P] =(MSet[P], Int, MSet[P], MSet[P])

  override def toPartialFunction[P](pn: PetriNet[P]):
                        PartialFunction[MSet[P],Set[MSet[P]]] =
  {case m =>
        for ((cond, p, eff,inh) <- pn;
            //se non vi sono archi inibitori per la transizione
            if m disjoined inh;
            //se non esiste una transizione con più alta priorità
            if !pn.exists(tr => m.matches(tr._1) && !tr._1.matches(cond) && tr._2 > p);
            out <- m extract cond
            ) yield out union eff
  }
}
object PriorityPetriNet extends PetriNet with Priority
```

The algorithm is a *partial function* starting from a system state and then

1. for all the transitions
   a) verifies that there are no inhibitory arcs
   b) verifies that there is no possible transition from another place with higher priority
   c) extracts the starting place of the transition from the global state of the system (remove a token in the place from which the transition starts)
   d) yield the new state of the system (place - startTransitionPlace + arriveTransitionPlace)

During modeling it was decided not to consider the transitions that come out of the same place by priority, because the transition that has the highest priority will always be the only one performed. Therefore, in this specific case, I have chosen to maintain nondeterminism so that the priorities have an effect on the order of the transitions, but not on their firing.

1.4.2 Example

Here is an example of a Petri Net with the Priority extension.

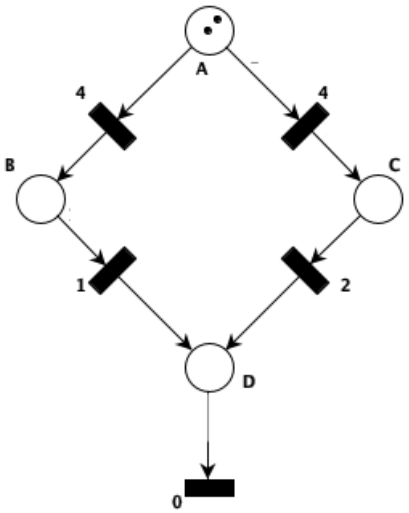

Figure 6: Example of a petri net with priorities

```
def prioritySystem() = PriorityPetriNet.toSystem(PriorityPetriNet[Place](
    (MSet(A), 4, MSet(B), MSet()),
    (MSet(A), 4, MSet(C), MSet()),
    (MSet(B), 1, MSet(D), MSet()),
```

```
  (MSet(C), 2, MSet(D), MSet()),
  (MSet(D), 0, MSet(), MSet()))
)
```

The following are the paths obtained for the Petri Net, with and without priorities.

```
Classic
List({A|A}, {B|A}, {D|A}, {C|D}, {C}, {D}, {})
List({A|A}, {B|A}, {D|A}, {C|D}, {D|D}, {D}, {})
List({A|A}, {B|A}, {D|A}, {A}, {C}, {D}, {})
List({A|A}, {B|A}, {D|A}, {A}, {B}, {D}, {})
List({A|A}, {B|A}, {D|A}, {B|D}, {B}, {D}, {})
List({A|A}, {B|A}, {D|A}, {B|D}, {D|D}, {D}, {})
List({A|A}, {B|A}, {B|B}, {B|D}, {D|D}, {D}, {})
List({A|A}, {B|A}, {B|B}, {B|D}, {B}, {D}, {})
List({A|A}, {B|A}, {C|B}, {C|D}, {D|D}, {D}, {})
List({A|A}, {B|A}, {C|B}, {C|D}, {C}, {D}, {})
List({A|A}, {B|A}, {C|B}, {B|D}, {B}, {D}, {})
List({A|A}, {B|A}, {C|B}, {B|D}, {D|D}, {D}, {})
List({A|A}, {C|A}, {C|C}, {C|D}, {D|D}, {D}, {})
List({A|A}, {C|A}, {C|C}, {C|D}, {C}, {D}, {})
List({A|A}, {C|A}, {C|B}, {C|D}, {C}, {D}, {})
List({A|A}, {C|A}, {C|B}, {C|D}, {D|D}, {D}, {})
List({A|A}, {C|A}, {C|B}, {B|D}, {B}, {D}, {})
List({A|A}, {C|A}, {C|B}, {B|D}, {D|D}, {D}, {})
List({A|A}, {C|A}, {D|A}, {A}, {B}, {D}, {})
List({A|A}, {C|A}, {D|A}, {A}, {C}, {D}, {})
List({A|A}, {C|A}, {D|A}, {B|D}, {D|D}, {D}, {})
List({A|A}, {C|A}, {D|A}, {B|D}, {B}, {D}, {})
List({A|A}, {C|A}, {D|A}, {C|D}, {D|D}, {D}, {})
List({A|A}, {C|A}, {D|A}, {C|D}, {C}, {D}, {})
```

```
With priority
List({A|A}, {B|A}, {B|B}, {B|D}, {D|D}, {D}, {})
List({A|A}, {B|A}, {C|B}, {B|D}, {D|D}, {D}, {})
List({A|A}, {C|A}, {C|C}, {C|D}, {D|D}, {D}, {})
List({A|A}, {C|A}, {C|B}, {B|D}, {D|D}, {D}, {})
```

Many of the possible paths in a "classic" scenario are cut off by priorities, because only the transitions with the highest priority among the possible ones can be fired.

In this specific case, tokens are "collected" in places B and C, until they are exhausted in place A. At this point, the tokens in place C have precedence over tokens in B, because the transition has a higher priority. The tokens are collected again in place D and, once exhausted in B and C, they will be consumed by the last transition.

2 Lab 03

2.1 Models analysis and properties verification

A model is an abstraction (and simplification) of a complex system (reality). The creation of a model that captures the relevant elements of a system is useful in order to do simulations and checks for the system itself.

Propositional Logic The simplest logic for property verification is called Propositional Logic. It allows to express static properties, such as properties that are initially valid in the system. If a proposition hold or not doesn't depend on any concept of system evolution.

Computational Tree Logic **CTL**, on the other hand, allows to express concepts of temporality and to verify **liveness** and **safety** properties during the execution of a system.

$\phi = p \mid T \mid F \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid A\psi \mid E\psi$

$\psi = X\phi \mid G\phi \mid F\phi \mid \phi U \phi$

In this logic, ϕ represents a property in a certain state, while ψ represents a property along a path.

PCTL Considering DTMC and the fact that some models may have infinite paths (with finite states), using operators like A (all paths) and E (at least one path) of **CTL** is not useful. This is because in this type of systems the solutions tend to converge. It becomes much more meaningful to verify properties within a certain number of steps.

Instead of A and E operators, have a P_J operator, where J is an interval into $[0, 1]$. $P_J\psi$ indicates that the probability of a certain path ψ occurrs has probability J.

The U operator is defined *bounded* and limits the number of steps within which something should happen. Starting from $U^{<=n}$, also $G^{<=n}$ and $F^{<=n}$ are derived.

$\phi = p \mid T \mid F \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid P_J(\psi)$

$\psi = X\phi \mid G^{\leq n}\phi \mid F^{\leq n}\phi \mid \phi U^{\leq n}\phi$

CSL CTMC doesn't consider a discrete time but a continuous one. We no longer talk about steps, and markovian rates no longer represent probabilities but

- Normalized probabilities
- Timed transitions

The logic used for these systems is **CSL - Continuous Stochastic Logic**. This is very similar to PCTL, but this logic uses real numbers instead of integers.

One more operator is $S_J(\phi)$, called **steady-state**. The operator applies to systems that never terminate and estimates, through J, how long the system remains in a certain state ϕ.

$\phi = p \mid T \mid F \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid P_J(\psi) \mid S_J(\phi)$

$\psi = X\phi \mid G^{\leq n}\phi \mid F^{\leq n}\phi \mid \phi U^{\leq n}\phi$

2.2 The PRISM curious

- Make the stochastic Readers & Writers Petri Net seen in lesson work: perform experiments to investigate the probability that something good happens within a bound
- Play with PRISM configuration to inspect steady-state proabilities of reading and writing (may need to play with options and choose "linear equations method")

The following Petri Net describes the problem of readers and writers and it's defined **stochastic** because the transitions have *Markovian rates*, which express properties of temporality and probability.

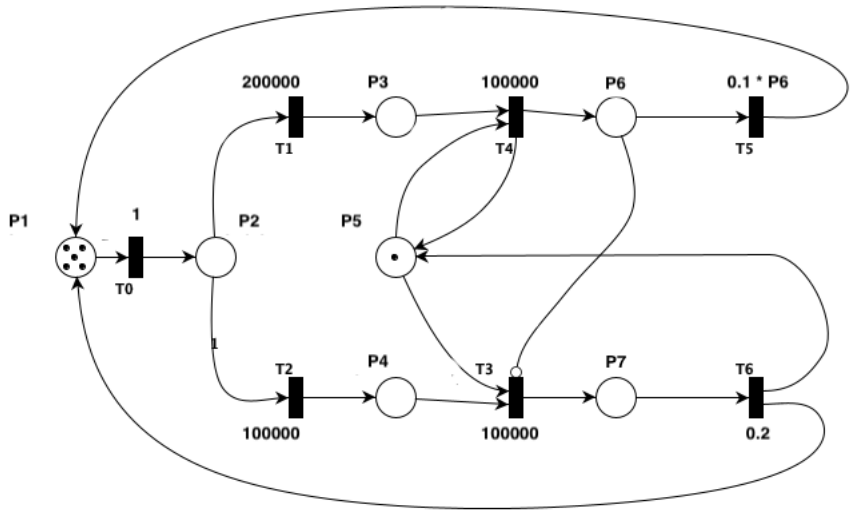

Figure 7: Stochastic Petri Net

Bounded / Unbounded Petri Net This network is called **bounded** because it has a finite number of states and the number of tokens within the network remains constant over time. However, the network has infinite *paths*, that is, paths that never end, but the states crossed along it will always be the same (no new scenarios are generated), because the number of tokens remains constant.

2.2.1 Read and write considerations

Probability of reading/writing with a bound With the following experiment I tried to quantify the probability of performing a reading or writing within a bound, comparing and evaluating the transitions and the markovian rates defined for them.

The properties are as follows

- $P_{[?]}$ [$(true)\ U^{\leq k}\ (p7 > 0)$] equivalent to $P_{[?]}$ [$F^{\leq k}\ (p7 > 0)$]
- $P_{[?]}$ [$(true)\ U^{\leq k}\ (p6 > 0)$] equivalent to $P_{[?]}$ [$F^{\leq k}\ (p6 > 0)$]

which represent in order

- probability that a process **reads** within k time unit
- probability that a process **writes** within k time unit

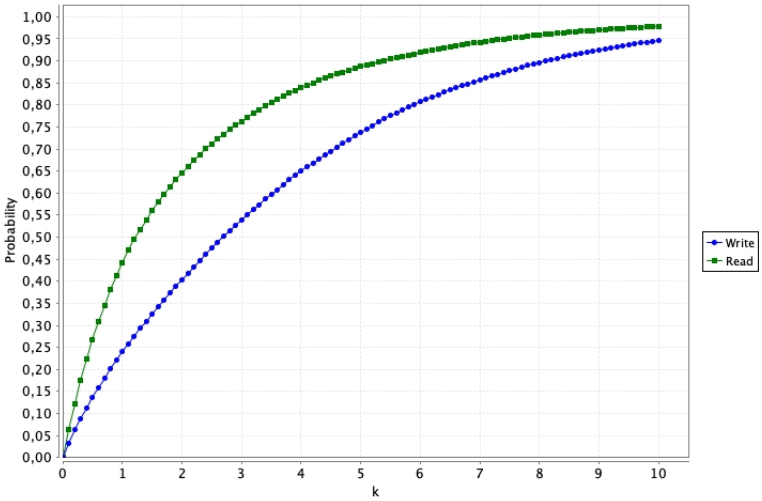

Figure 8: Probability that a process read or write within k time unit

The graph shows the different probabilities in terms of reading and writing. The different curves are due to the different markovian rates, both at the level of normalized probabilities and at the level of transitions execution time.

In the first case, the markovian rates on T1 and T2 are one double the other. Therefore, at the exit from P2, there will be about 66% probability of arriving in P3 and about 33% probability of arriving in P4 (values obtained by normalizing the markovian rates starting from P2). This Markovian rate also means that T1 (ASK READ) fires twice as fast as T2 (ASK WRITE).

Another aspect to consider about the curves is the relatively high read and write times ($0.1 * p6$ and 0.2).

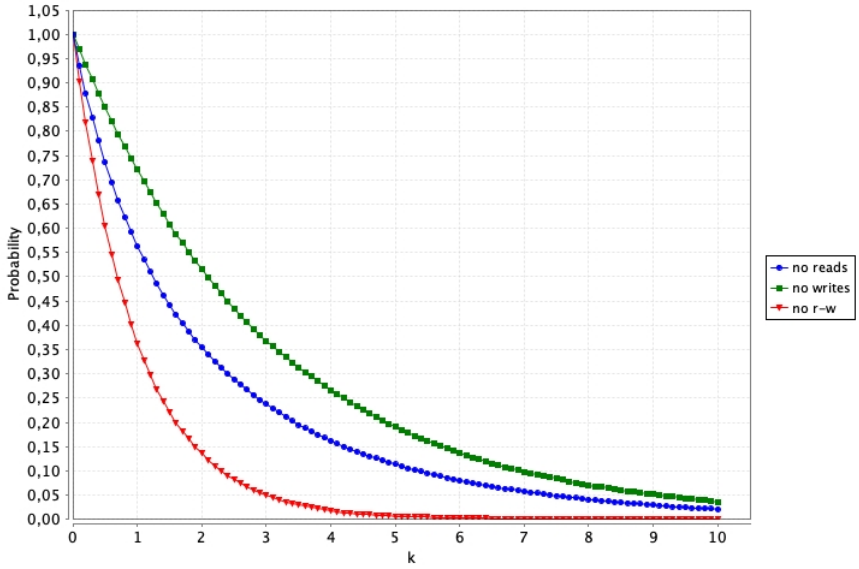

Figure 9: Probability that no reads or writes occur in k steps

Probability that a read or write will not occur This verification is the opposite of the previous one, I try to understand the probability that no reads or writes occur in k time unit.

The defined properties are as follows

- $P_{[?]}$ [$G^{\leq k}$ $\neg(p6 > 0)$]
- $P_{[?]}$ [$G^{\leq k}$ $\neg(p7 > 0)$]
- $P_{[?]}$ [$G^{\leq k}$ $\neg(p6 > 0)$ $\wedge$ $G^{\leq k}$ $\neg(p7 > 0)$]

2.2.2 Steady-state

The $S_J(\phi)$ operator applies to systems that never terminate and estimates, through J, how long the system stays in a certain ϕ state.

The checks are performed with the *Backwards Gauss-Seidel* resolution method.

- 86% of the time the system has at least one process that reads or writes

  $S_{[?]}$ [$(p7 > 0) \vee (p6 > 0)$]
- 52% of the time the system has a writing process

  $S_{[?]}$ [$(p7 > 0)$]
- 34% of the time the system has at least one process reading

  $S_{[?]}$ [$(p6 > 0)$]

- 52% of the time the permit (token in P5) is taken by some process

  $S_{[?]}\ [\ (p5 = 0)\ ]$

- 64% of the time there are processes waiting to read or write

  $S_{[?]}\ [\ (p3 > 0) \vee (p4 > 0)\ ]$

- 53% of the time there's at least one process waiting to write while another process is doing it or other processes are reading

  $S_{[?]}\ [\ (p4 > 0) \wedge ((p6 > 0) \vee (p7 > 0))\ ]$

2.2.3 Rewards

I tried to experiment with PRISM'S *reward* function. It's possible, when a certain event occurs (such as the firing of a transition), to assign rewards. In this case, two are defined, one for writing and one for reading.

```
rewards "reads"
    [t4] true : 1;
endrewards
```

```
rewards "writes"
    [t5] true : 1;
endrewards
```

I tried to understand, through the C property (*cumulative reward*), an estimate on the number of reads and writes expected after t time unit.

The property $C_{\leq t}$ corresponds to the reward accumulated along a path after time t. It can also be used in DTMC, where t is defined with an integer and represents the number of *steps*.

Reads reward number after t time unit $R_{reads=?}[C_{\leq t}]$

Writes reward number after t time unit $R_{writes=?}[C_{\leq t}]$

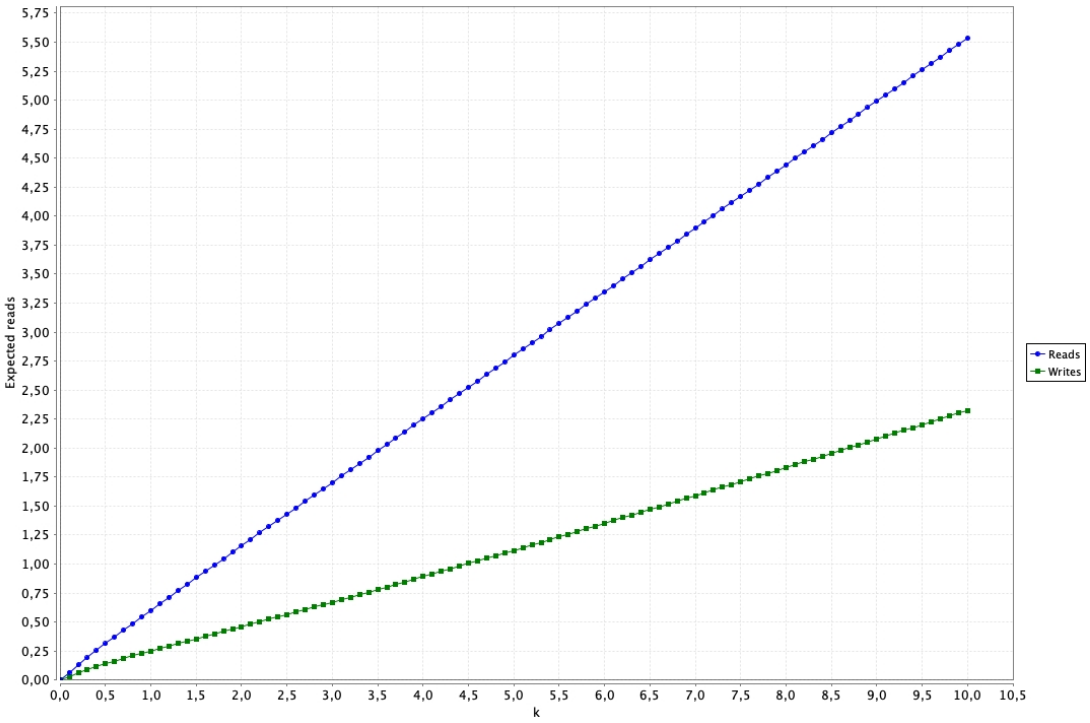

Figure 10: Expected reads and writes after t time units

2.3 The PRISM vs Scala investigator

- Take the communication channel example, and perform comparison of results between PRISM and our Scala approach
- Write Scala support for performing additional experiments and comparisons (e.g., G formulas, steady-state computations)

The *CTMCAnalysis* source allows to perform *experiment*, as intended on PRISM. An experiment is a set of verifications in which a parameter is changed in each verification in order to understand how this affects the system. Usually the variable is time, so I try to understand the probability in which something good happens before a given time t.

Type Property, in the Scala model, is a function that goes from a *trace* (path of a run) to a Boolean.

```
type Property[A] = Trace[A] => Boolean
```

2.3.1 New operators

Globally $G\ \phi$ The *globally* operator is obtained starting from the *eventually* operator by equivalence $\neg G\phi \equiv F\neg\phi$

In fact, ϕ is always true (*globally*) if it isn't true that somewhere (*eventually*) inside the trace ϕ is false.

```
def globally[A](filt: A => Boolean): Property[A] =
    (trace) => !eventually((s:A) => !filt(s))(trace)
```

Until $\phi\ U\ \phi'$ is true if ϕ' is true in a given time t_n and ϕ is true from t_0 to t_{n-1}.

In other words, $\phi\ U\ \phi'$ is true if ϕ hold at least until ϕ' becomes true.

The function takes two conditions, ϕ and ϕ', named *first* and *second* respectively.

```
def until[A](first: A => Boolean, second: A => Boolean): Property[A] =
    (trace) => {
      trace.exists{case (t,a) => second(a)} &&
        trace.takeWhile{case (t,a) => !second(a)}.forall{case (t,a) => first(a)}
    }
```

The first condition is used to check whether ϕ' occurs within the trace. The second condition checks that ϕ hold, as long as ϕ' doesn't hold.

2.3.2 PRISM vs Scala

The model to be checked is shown in figure 11.

Properties to check

- $P_? \, [\, true \, U^{\leq k} \, S_3 \,] \equiv P_? \, [\, F^{\leq k} \, S_3 \,]$
- $P_? \, [\, G^{\leq k} \, \neg \, S_3 \,]$

PRISM declaration

```
P=? [ (true) U<=k (s=3) ]
P=? [ F<=k (s=3) ]
P=? [ G<=k (!(s=3)) ]
```

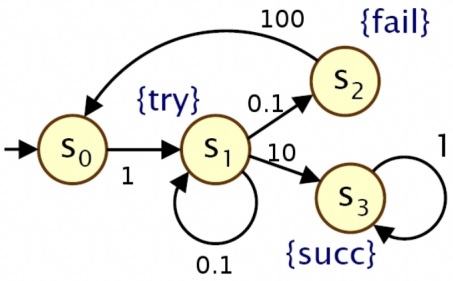

Figure 11: CTMC communication channel

Scala declaration

```
val untilProp = channelAnalysis.until[State](_ => true, _ == DONE)
val eventuallyProp = channelAnalysis.eventually[State](_ == DONE)
val globallyProp = channelAnalysis.globally[State](_ != DONE)
```

Same experiments in Scala and in PRISM, with k from 0 to 10 considering 0.1 steps. Below are the graphs obtained for the first two properties.

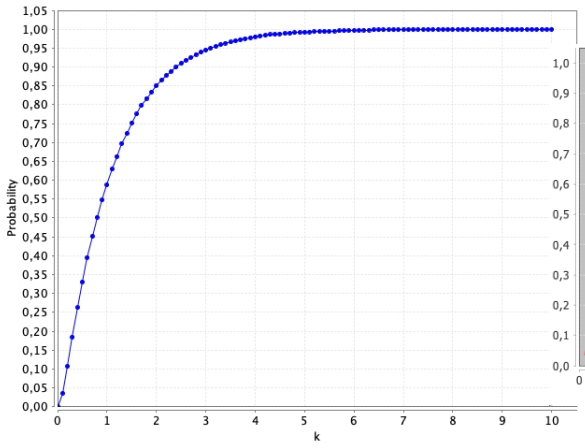

Figure 12: Experiment with PRISM

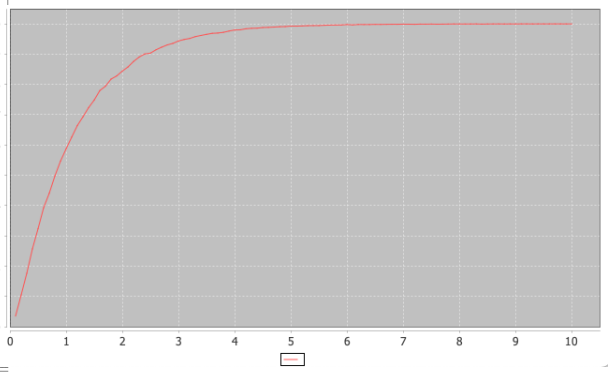

Figure 13: Experiment with Scala

3 Lab 05

3.1 Aggregate Computing

Aggregate computing is a research thread that tries to find nice abstractions and a good programming language for macro/collective programming. In this scenario, we want to program not the single device, but the collective behavior for the aggregation of devices.

3.1.1 Spatial streams

A key abstraction is called *spatial stream*. A spatial stream is not just a sequence of data, but a sequence of data distributed in space. In fact, in a context of aggregate computing there isn't a single stream, but several streams coming from a multitude of devices distributed within a cluster. We try to express behavior, even complex ones, by manipulating these spatial streams.

3.1.2 Computational fields

The idea is that a field is a map from a point where there is a device in space to a computational value (number, boolean, complex data structure).

The *space-value* mapping is simplistic, because the time component is not being considered. In fact, time must necessarily be considered in non-static situations. Unless the devices within a system are static sensors, their position will change over time.

A *field* is considered to be a structure concerning space and time. The domain is a set of *events* produced by devices. An event is a value produced by a device in a certain position and at a certain moment in time. The devices don't produce events in the same way and at the same time, as in CTMC, they could produce events at different times from each other.

3.1.3 Lab activities

Case 9: experimenting with branch and mux

- where sense1 is active count from 0 to 1000 and then stay freezed at 1000, otherwise 0

→ use mux externally, see what happens with multiple clicks, and then use branch instead

mux is a ternary operator, which evaluates a condition and executes one of the two scenarios specified in the input. *branch* has the same syntax, but divides the computation into two distinct spaces. This creates several regions in which the computations are performed.

```
mux(sense1)(rep(0){x => mux(x <1000)(x+1)(x)})(0)
branch(sense1)(rep(0){x => mux(x <1000)(x+1)(x)})(0)
```

Running *mux*, activating *sense1* on one or more devices, will have no effect. The device will immediately take value 1000 because the computation started immediately after the declaration (this scenario could be defined *eager*).

Using *branch* instead, each time, the count from 0 to 1000 is always performed.

At the *engine* level, *branch* is built from *mux*, but the parameters are passed by *call by name*: this *evaluation strategy* can be defined *lazy*.

```
def mux[A](cond: Boolean)(th: A)(el: A): A = if (cond) th else el

def branch[A](cond: => Boolean)(th: => A)(el: => A): A =
      mux(cond)(() => aggregate{ th })(() => aggregate{ el })()
```

Case 12: experimenting with Scala interoperability, return a Set[Int]

- gather in each node the set of neighbours' IDs..

→ used foldhood, help type inference (it's a Set[ID])

Exploiting the *foldhood* operator, the computations deriving from the neighbors of each device are combined.

```
def main():Set[Int] = foldhood(Set[ID]())(_ ++ _)(nbr{Set(mid())})
```

nbr is a function computed locally by the device and sent by it to all its neighbors.

Case 8: experiment with pairs

- have in each node the ID of the closest neighbour

→ used minHoodPlus, construct a pair of distance (nbrRange) and id (idnbrm), note minHoodPlus correctly works on pairs

Using the *minHoodPlus* operator I find, among all the neighbors of a device, the nearest.

```
def main():(Int, ID) = minHoodPlus(nbrRange.toInt, nbr{mid()})
```

Case 14: understand gossiping

- gossip the maximum value of ID (type Int)
- note a problem: it won't correctly repair upon network changes

→ use max and maxHoodPlus smoothly

Using the *rep* operator, I create a *repetition*, that is an infinite stream of data starting from an initial value and a function that specifies how the sequence proceeds.

```
def main(): ID = rep(mid()){x => x max maxHoodPlus(nbr(x)) }
```

In this way it's possible to propagate, through gossip, the maximum value among the IDs within the device's network. The value from which all devices start is their *mid*, but this is replaced if there is a neighbor with a larger ID. The largest value is then propagated to the neighbors.

Following tests, it has been noted that the network is unable to update itself after changes. A possible solution was found in evaluating not the *x*, but the *mid*.

```
def main(): ID = rep(mid()){x => mid() max maxHoodPlus(nbr(x)) }
```

Case 16

- define a gradient that stretches distances so that where sense2 is true they become 5 times larger

→ tweak your usage of nbrRange

The development of this exercise is based on the example presented in class, in the Main16 class of the Programs.scala source

```
class Main16 extends AggregateProgramSkeleton {
  override def main() = rep(Double.MaxValue){ d => mux[Double](sense1)
                                                   {0.0}
                                                   {minHoodPlus(nbr{d}+nbrRange)}
  }
}
```

In this example, a gradient is calculated starting from a source. This represents the minimum distance that separates the source from other devices.

If a device has *sense2* active, then its gradient will increase by 5 times. This will affect all devices whose minimum path passed through it.

However, the gradient of these devices is not increased by 5 times, but only by the increase that the device with *sense2* true has undergone. However, if there is an alternative path that doesn't pass through this device, then the gradient will be calculated on this one. In this case there will be an increase in the gradient equal to the difference between the old shortest path and the new shortest path.

```
def gradient(src: Boolean)(stretch: Boolean)(factor: Int): Double =
      rep(Double.MaxValue){
        d => mux(src)
                {0.0}
```

```
                {minHoodPlus((nbr{d} + nbrRange) * mux(stretch)(factor)(1))}
    }

override def main() = gradient(sense1)(sense2)(5)
```

A new block: partition

- propagate a pair instead of a double as in case 8
- first component is distance as usual
- second component is a source's mid

→ minHoodPlus already works for pairs

→ use, nbr of first component and second component separately

A tuple is propagated (range, sourceMid) instead of just the double with the distance between two devices. Partitions are created and highlighted by the second element of the tuple on each device. Hence this will be the ID of the partition (and of the source device)

```
def distanceFromSource(d: (Double, ID)) =
    minHoodPlus(nbr{d._1} + nbrRange, nbr{d._2})

def partition(src:Boolean, mid: ID): (Double, ID) =
    rep(Double.MaxValue, mid){ds => mux(src){(0.0, mid)}{distanceFromSource(ds)}}

override def main(): (Double, ID) = partition(sense1, mid())
```

To better highlight partitions, it's possible to return only the second element of the tuple (._2)

```
def partition(src:Boolean, mid: ID): ID =
    rep(Double.MaxValue, mid){ds => mux(src){(0.0, mid)}{distanceFromSource(ds)}}._2
```

Do channel

- Step 0
  - created gradient by case 16, simply with one input source
- Step 1
  - code broadcast: it is just a variation of partition, really
  - inputs: source field, input field (it was mid for partition)
  - output: the result of broadcasting input field outward sources
- Step 2
  - use broadcast to define block distance
  - distance is achieved by broadcast the result of gradient at destination
- Step 3
  - you are ready for the channel

Gradient calculates the gradient, which is the distance from one point to another. It takes a boolean (source device) as input and calculates the distance from it to all other devices.

```
def gradient(src:Boolean): Double =
    rep(Double.MaxValue){d => mux[Double](src){0.0}{minHoodPlus(nbr{d} + nbrRange)}}
```

Broadcast is a variation of partition. Partition propagated a tuple using *rep*. It takes a boolean (source) and a double as input. Broadcast also spread the value to all devices.

```
def broadcast(src: Boolean, input: Double): Double = {
    rep((Double.MaxValue, input)){ dv =>
        mux(src){(0.0, input)}{minHoodPlus((nbr{dv}._1 + nbrRange, nbr{dv}._2))}
    }._2
  }
```

Distance Broadcast the distance between two devices, identified through two source sensors.

```
def distance(src: Boolean, dst: Boolean): Double = {
    broadcast(src, gradient(dst))
  }
```

Channel starting from two Booleans (source and destination) it highlights to *true* the devices that are part of a minimum path from the source device to the destination device.

```
def channel(src: Boolean, dst: Boolean) =
    gradient(src) + gradient(dst) <= distance(src, dst)
```

Initially, the distance of a point from a source device and a destination device is calculated. If the sum of these two values is less than or equal to the distance between *src* and *dst* then the device is part of the shortest path. The algorithm works because the shortest path (the *distance* method) is calculated using the same principle.