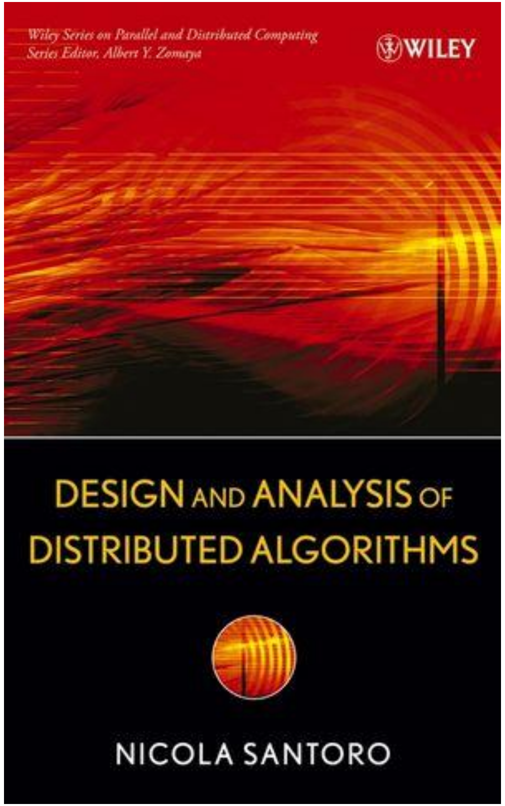
Towards an abstract syntax for distributed algorithms

Presentation by Rúben Guilherme, Paulo Duarte and João Fraga

The book

- Very simple and understandable.
- Very good specification of what a distributed environment is.
- Easy to read pseudo-code for the algorithms which makes it easier for us to create a simple implementation.



The Flooding example - book

PROTOCOL Flooding .

- Status Values: $\mathcal{S} = \{\text{INITIATOR, IDLE, DONE}\}$;
  $\mathcal{S}_{\text{INIT}} = \{\text{INITIATOR, IDLE}\}$;
  $\mathcal{S}_{\text{TERM}} = \{\text{DONE}\}$.
- Restrictions: Bidirectional Links, Total Reliability, Connectivity, and Unique Initiator.

```
INITIATOR
    Spontaneously
    begin
        send(M) to N(x);
        become DONE;
    end

IDLE
    Receiving(I)
    begin
        Process(M);
        send(M) to N(x) - {sender};
        become DONE;
    end
```

The Flooding example - syntax part 1

- First the user defines the states of the entities.
- Then, the restrictions this protocol uses.
- Afterwards, the behaviour of the entities during the protocol.

```
section protocol Flooding {
  state = [INITIATOR, IDLE, DONE];
  init = [INITIATOR, IDLE];
  term = [DONE];

  restric = [BidirectionalLinks, TotalReliability, Connectivity, UniqueInitiator];

  behaviour = {
    INITIATOR, spontaneously,
    begin
      send(M) to neighbours();
      become DONE;
    end

    IDLE, Receiving(M),
    begin
      PROCESS(M);
      send(M) to neighbours() - {sender};
      become DONE;
    end
  }

  procedures = {
    PROCESS
    begin
      (* nothing to process *)
    end
  }
}
```

The Flooding example - syntax part 2

We also decided to define the network on a different section of the code.

```
section network {
  entities = [
    1, INITIATOR;
    2, IDLE;
    3, IDLE;
    4, IDLE;
    5, IDLE;
  ]

  edges = [
    1, 2;
    1, 3;
    2, 3;
    2, 4;
    2, 5;
    3, 4;
    3, 5;
    4, 5;
  ]
}
```

AST Implementation

From code

To trees

Entity

```
type 'a register = 'a array

type 'a memory = (string, 'a) Hashtbl.t

type state = string

type entity = {
  id : int;
  clock : int;
  register : int register;
  memory : int memory;
  state : state;
}
```

Capabilities Each entity $x \in \mathcal{E}$ is endowed with local (i.e., private and nonshared) memory M_x. The *capabilities* of x include access (storage and retrieval) to local memory, local processing, and communication (preparation, transmission, and reception of messages). Local memory includes a set of *defined registers* whose values are always initially defined; among them are the *status register* (denoted by $status(x)$) and the *input value register* (denoted by $value(x)$). The register $status(x)$ takes values from a finite set of system states $\mathcal{S}$; the examples of such values are "Idle," "Processing," "Waiting,"... and so forth.

In addition, each entity $x \in \mathcal{E}$ has available a local *alarm clock* c_x which it can set and reset (turn off).

An entity can perform only four types of *operations*:

- local storage and processing
- transmission of messages
- (re)setting of the alarm clock
- changing the value of the status register

External event

```
type event = MessageReceived of message | AlarmRing | Spontaneous
```

External Events The behavior of an entity $x \in \mathcal{E}$ is *reactive*: x only responds to external stimuli, which we call *external events* (or just *events*); in the absence of stimuli, x is inert and does nothing. There are three possible external events:

- arrival of a message
- ringing of the alarm clock
- spontaneous impulse

Message

```
type message = {
  sender : entity;
  receiver : entity;
  message : string;
  code : int;
}
```

Behaviour

```
type behaviour = (state * event, statement) Hashtbl.t
```

Behavior The nature of the action performed by the entity depends on the nature of the event e, as well as on which status the entity is in (i.e., the value of $status(x)$) when the events occur. Thus the specification will take the form

$$\text{Status} \times \text{Event} \longrightarrow \text{Action},$$

Restrictions

```
type restriction =
  | MessageOrdering
  | ReciprocalCommunication
  | BidirectionalLink
  | EdgeFailureDetection
  | EntityFailureDetection
  | GuaranteedDelivery
  | PartialReliability
  | TotalReliability
  | Connectivity
  | BoundedCommunicationDelays
  | UnitaryCommunicationDelays
  | SynchronizedClocks
  | UniqueInitiator
```

- *Message Ordering:* In the absence of failure, the messages transmitted by an entity to the same out-neighbor will arrive in the same order they are sent.
- *Reciprocal communication:* $\forall x \in \mathcal{E}, N_{\text{in}}(x) = N_{\text{out}}(x)$. In other words, if $(x, y) \in \vec{E}$ then also $(y, x) \in \vec{E}$.
- *Bidirectional links:* $\forall x \in \mathcal{E}, N_{\text{in}}(x) = N_{\text{out}}(x)$ **and** $\lambda_x(x, y) = \lambda_x(y, x)$.
- *Edge failure detection:* $\forall (x, y) \in \vec{E}$, both x and y will detect whether (x, y) has failed and, following its failure, whether it has been reactivated.
- *Entity failure detection:* $\forall x \in V$, all in- and out-neighbors of x can detect whether x has failed and, following its failure, whether it has recovered.
- *Guaranteed delivery:* Any message that is sent will be received with its content uncorrupted.
- *Partial reliability:* No failures will occur.
- *Total reliability:* Neither have any failures occurred nor will they occur.
- *Connectivity:* The communication topology $\vec{G}$ is strongly connected.
- *Bounded communication delays:* There exists a constant Δ such that, in the absence of failures, the communication delay of any message on any link is at most Δ.
- *Synchronized clocks:* All local clocks are incremented by one unit simultaneously and the interval of time between successive increments is constant.
- *Unitary communication delays:* In the absence of failures, the communication delay of any message on any link is one unit of time.

4. Unique Initiator, that is, only one entity will start.

Protocol

```
type protocol = {
  name : string;
  state : state list;
  init : state list;
  term : state list;
  restric : (restriction, unit) Hashtbl.t;
  behaviour : behaviour;
  procedures : statement list;
}
```

Network

```
type network = (edge * edge, unit) Hashtbl.t
```

```
type edge = entity * entity
```

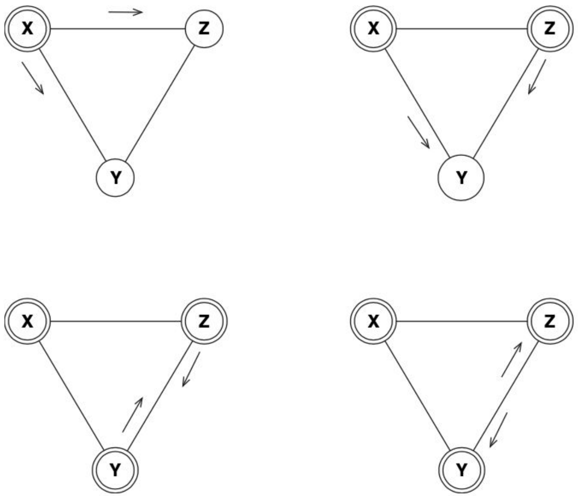


FIGURE 1.3: An execution of Flooding.

Program

```
type program = { protocol : protocol; network : network }
```

General Programming

```
type identifier = string

type uniop = Uneg | Unot

type binop =
    | Band
    | Bor
    | Bxor
    | Badd
    | Bsub
    | Bmul
    | Bdiv
    | Bmod
    | Beq
    | Bne
    | Blt
    | Bgt
    | Ble
    | Bge
```

```
type constant =
  | Cbool of bool
  | Cint of int
  | Cfloat of float
  | Cstring of string

type expression =
  | Econst of constant
  | Evar of identifier
  | Epair of expression * expression
  | Euniop of uniop * expression
  | Ebinop of binop * expression * expression

type statement =
  | Sblock of statement list
  | Sexp of expression
  | Satr of identifier * expression
  | Sif of expression * statement * statement
  | Sspontaneously of statement
  | Srecieve of event
  | Ssend of event
  | Sbecome of state
  | Sneighbours of entity
```

Overview & future work

```
section protocol Flooding {
  state = [INITIATOR, IDLE, DONE];
  init = [INITIATOR, IDLE];
  term = [DONE];

  restric = [BidirectionalLinks, TotalReliability, Connectivity, UniqueInitiator];

  behaviour = {
    INITIATOR, spontaneously,
    begin
      send(M) to neighbours();
      become DONE;
    end

    IDLE, Receiving(M),
    begin
      PROCESS(M);
      send(M) to neighbours() - {sender};
      become DONE;
    end
  }

  procedures = {
    PROCESS
    begin
      (* nothing to process *)
    end
  }
}
```

```
(* entity *)
type 'a register = 'a array

type 'a memory = (string, 'a) Hashtbl.t

type state = string

type entity = {
  id : int;
  clock : int;
  register : int register;
  memory : int memory;
  state : state;
}

(* event *)
type message = {
  sender : entity;
  reciever : entity;
  message : string;
  code : int;
}

type event = MessageReceived of message | AlarmRing | Spontaneous

(* general programming *)
type identifier = string

type uniop = Uneg | Unot

type binop =
  | Band
  | Bor
  | Bxor
  | Badd
  | Bsub
  | Bmul
  | Bdiv
  | Bmod
  | Beq
```