Integration of a modelling language and a distributed programming language

Alexis Grall

11/06/2019

advisors :
Dominique Mery and Horatiu Cirstea

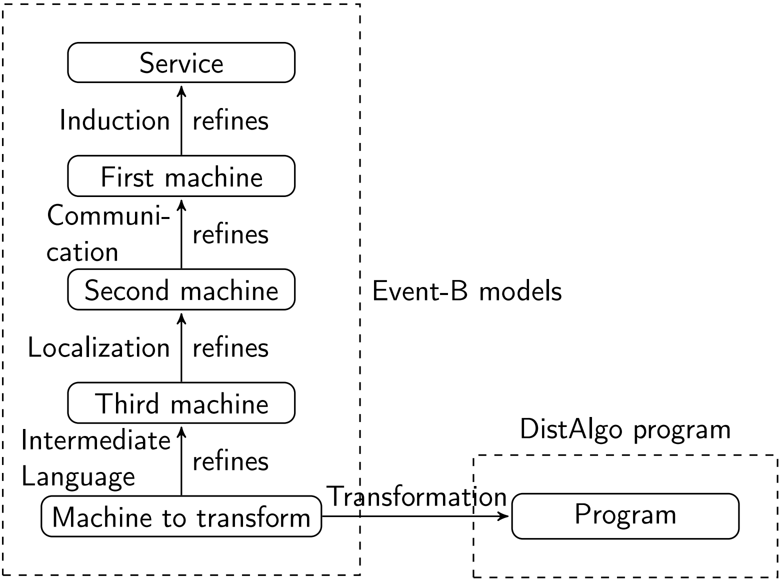
Introduction

Main objective : automatically translate models of distributed algorithms into executable programs.

The models are obtained by refinement with the Event-B method.

The chosen programming language is DistAlgo.

Introduction



Event-B

- Method for modelling systems
- Based on set theory
- Refinement of models
- Proof assistant and model checker.
- Framework Rodin is used.
- A model consists of contexts and machines

Context

A context specifies

- Types (which are sets)
- Constants
- Axioms and theorems

Machine

A machine specifies

- Variables
- Invariants on the variables
- Events which define the state transition of the system.

Events

An *INITIALISATION* event initialises the variables of the machine.
Other events are made of :

- Parameters
- Guards on the variables and parameters
- Actions which modify the variables

When there exists values for the parameters for which the guards are true, the event is then enabled.
When an event is enabled, the actions of the event can be observed.

DistAlgo

- Programming language for distributed algorithms
- High level
- Based on Python

Structure of a DistAlgo program

A DistAlgo program defines some class of processes and a main method.

The main method instantiates the processes of each class of process and runs them.

A class of processes consists of

- A setup method to initialise the local variables of the process.
- A run method with the program of the process.
- Message handlers that are executed when messages are received.
- User methods

Program structure

```
class A(process):
    def setup(args):
        setup_body
    def receive(msg=mexp, from_=pexp):
        receive_body
    def run():
        run_body
class B(process):
    ⋮
def main(*args):
    a = new(A)
    setup(a, (args))
    start(a)
```

Communication

Sending a message : `send(`*mexp*`, to=`*pexp*`)`

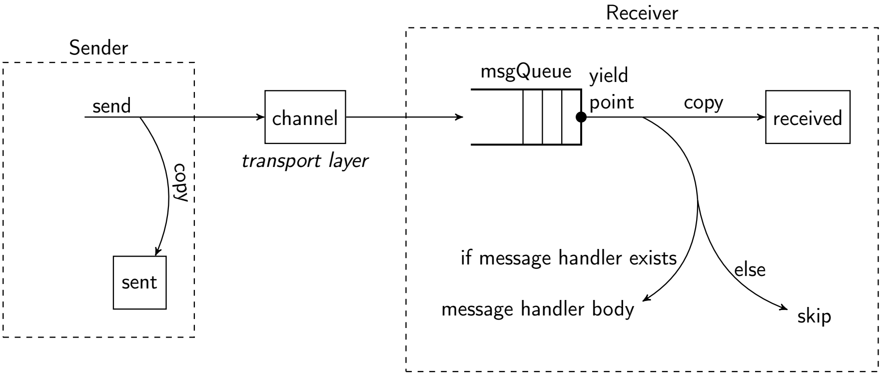
Messages are received at yield points : `await` statements.
The statement

`await(`*bexp*`)`

triggers the reception of all the messages that arrived before but were not yet received. The process then waits for the condition *bexp* to be true while new messages are received.

When a message is received, corresponding message handlers are executed.

Communication

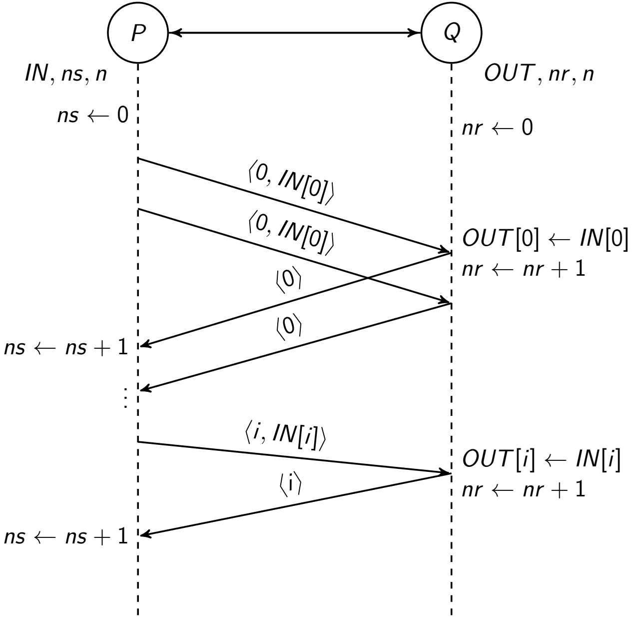


Stop-and-wait ARQ

A process p has an array IN of length n he wants to send to a process q and q wants to copy IN in an array OUT.

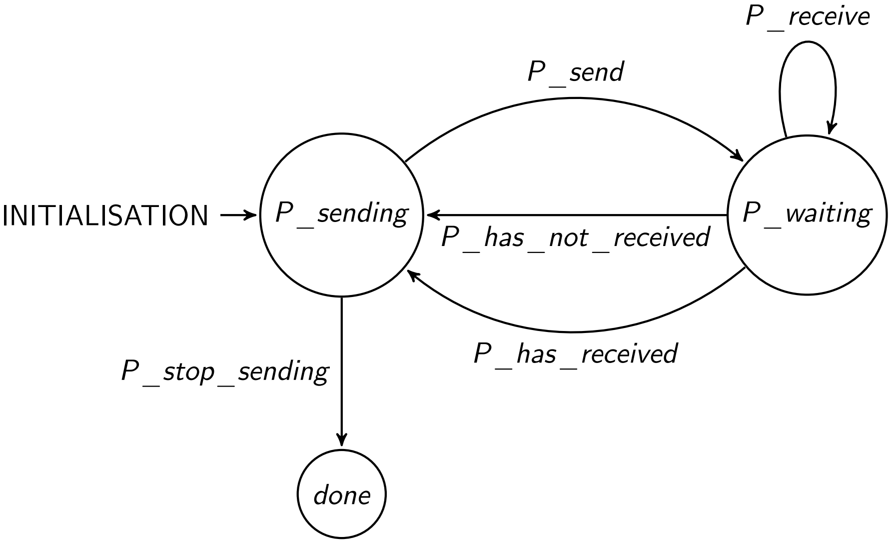
Loss of messages can happen.

Stop-and-wait ARQ



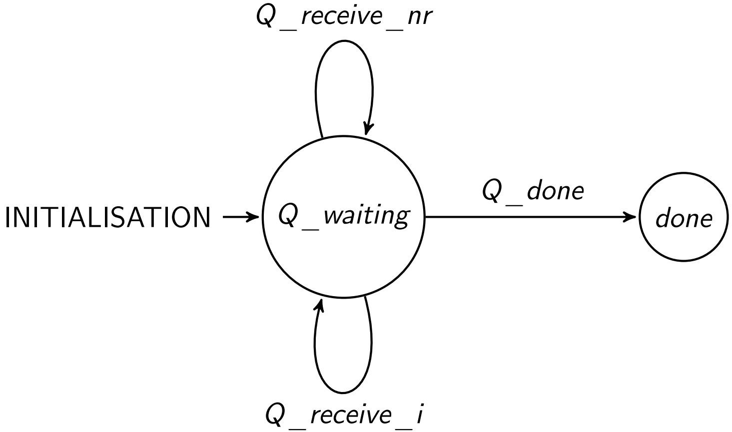
Automata for p

Here is the automata for the program of process p.



Automata for q

Here is the automata for the program of process q.



Context

```
CONTEXT C
SETS
    NODES MESSAGES MESSAGE_TYPES STATES
CONSTANTS
    P Q p q class of processes and processes
    P_sending P_waiting Q_waiting done reception_states states
    send receive lose functions for communication between processes
    IN n dest @P array to send, length of IN and dest = q
    source @Q source = p
AXIOMS
    ⋮
END
```

Transformation of the context

With the context, we can already get the main structure of the program.

```
class P(process):
    ⋮
class Q(process):
    ⋮
def main():
    p = new(P)
    q = new(Q)
    IN = ...
    n = ...
    setup(p, (IN, n, q))
    setup(q, (n, p))
    start({p,q})
```

Machine variables

OUT, ns and nr are variables only for this algorithm.

$channel$ is the variable modelling the communication channels.

pc is a function of $NODES \rightarrow STATES$ which gives the state of each process.

Initialisation

The initialisation of the different variables is the following.

MACHINE

EVENTS

Initialisation

then

act1: $OUT := \varnothing$ function with the empty set as domain

act2: $ns := 0$

act3: $nr := 0$

act4: $channel := emptyChannel$

act5: $pc := \{p \mapsto P\_sending, q \mapsto Q\_waiting\}$

END

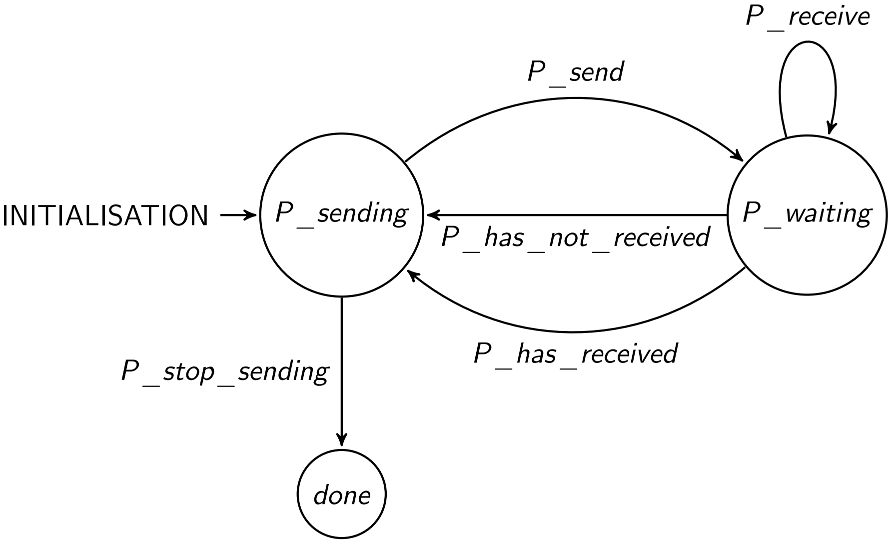
Setup of P

The setup method for class `P` is obtained from the Initialisation event.

```
class P(process):
    def setup(IN, n, dest):
        self.ns = 0
        self.pc = "P_sending"
    def run():
        ⋮
    def receive(...):
        ...
```

Events of p

The different events of p are the transitions of its automata.



Events of p

P_stop_sending is a local event of computation.

P_send is a sending event.

P_has_received is an await event.

P_has_not_received is an await event.

P_receive is a reception event.

Here, *P_waiting* is the only reception state. Events *P_has_not_received* and *P_has_received* are enabled in this reception state and thus are await events.

$P\_stop\_sending$

The event is enabled when *ns* is greater than the length of *IN*. It means that *p* has received all the needed acknowledgements from *q* and can stop sending data.

P_stop_sending $\widehat{=}$

when

grd1: $ns \geq n$

grd2: $pc(p) = P\_sending$

then

act1: $pc(p) := done$

P_send

The sending event is enabled when ns is less than the length of IN. In this case, p wants to send $IN[ns]$ to q.

P_send $\widehat{=}$

when

grd1: $pc(p) = P\_sending$

grd2: $ns < n$

then

act1: $channel := send(channel \mapsto (p \mapsto dest) \mapsto (data \mapsto data2msg(ns \mapsto IN(ns))))$

act2: $pc(p) := P\_waiting$

Await events

Event $P\_has\_received$ is an await event which is enabled when p has received an acknowledgement for the current value of ns.

P_has_received $\widehat{=}$
when
grd1: $pc(p) = P\_waiting$
grd2: $received(channel \mapsto (dest \mapsto p) \mapsto (ack \mapsto nat2msg(ns)))$
then
act1: $ns := ns + 1$
act2: $pc(p) := P\_sending$

Event $P\_has\_not\_received$ is similar but is enabled when p has not received an acknowledgement for the current value of ns and thus does not increment ns.

Reception event

The reception event is enabled when $pc(p)$ is in a reception state ($P\_waiting$ for this example) and a message is ready to be received by p.

P_receive $\widehat{=}$

any

s m

where

grd1: $pc(p) \in reception\_states$

grd2: $s \in NODES$

grd3: $m \in MESSAGE\_FUNCTIONS \times MESSAGES$

grd4: $readyToBeReceived(channel \mapsto (s \mapsto p) \mapsto (m))$

then

act1: $channel := receive(channel \mapsto (s \mapsto p) \mapsto m)$

Transformation of the machine

The different value of $pc(p)$ will corresponds to different methods of the class of p. The value of $pc(p)$ will determine which methods to execute.

```
class P(process):
    def setup(IN, n, dest):
        ...
    def PSending():
        ...
    def PWaiting():
        ...
    def run():
        state = {"P_sending":PSending, "P_waiting"
           :PWaiting}
        while(self.pc != "done"):
            states[self.pc]()
    def receive(...):
        ...
```

PSending

The two events that are enabled in the state *P_sending* are *P_send* and *P_stop_sending*. They are translated in the method PSending.

```
def PSending():
  # P_stop_sending
  if(self.ns>self.n):
      self.pc = "done"
  # P_send
  elif(self.ns<=self.n):
      send(('data', (self.ns, self.IN[self.
         ns])), to=self.dest)
      self.pc = "P_waiting"
```

PWaiting

The two events *P_has_received* and *P_has_not_received* are enabled in the state *P_waiting* which is a reception state. The events are therefore await events and translated in the PWaiting method.

```
def PWaiting():
  # P_has_received
  if await(received(('ack', self.ns), from_=
      self.dest):
       self.ns = self.ns+1
       self.pc = "P_sending"
  # P_has_not_received
  elif(not(received(('ack', self.ns), from_=
      self.dest)):
       self.pc = "P_sending"
```

Receive method

A reception event is enabled in a reception state and does not change the value of *pc*. It is translated by a receive method.

```
# P_receive
def receive(msg=(m), from_=s):
    pass
```

It does not do anything special apart from receiving the message in our case.

Conclusion

I showed with the example of the stop-and-wait algorithm how to transform a specific Event-B model of a distributed algorithm into a program.

The rules of this transformation are currently being defined on a sublanguage of Event-B.

The next goal will be to prove that this transformation is correct and to implement an automatic translation based on this transformation from Event-B to DistAlgo.

End slide

Questions ?