AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

Test of Distributed Systems Lecture 4

Data and Channels:

Data Structures

Program Structures

Channels

Today's lecture

- Arrays
- Records
- Symbolic Names
- Channels
- Channel variants
- Channel content

Array Syntax

- int a[5]
  is an array of int-numbers with five fields:
  a[0] to a[4]
- Arrays are always one-dimensional.
  You cannot write int a[5][2].
- Arrays are initialised field by field:
  a[0] = 2; a[1] = 3; a[2] = 5; a[3] = 7; a[4] = 11

Adding numbers

```
int a[5];
int x = 0;

active proctype P() {
 int i=0;
 a[0] = 2; a[1] = 3; a[2] = 5; a[3] = 7; a[4] = 11;
 do
 :: i>4 -> break
 :: else ->
   x = x + a[i];
   i++
 od;
 assert x == 28
}
```

Records

- Typedef MESSAGE {
- mtype message;
- byte source;
- byte dest;
- bool urgent
- }

- MESSAGE m;

- init {
- m.message = NUL;
- m.source = 1;
- m.dest = 2;
- m.urgent = false
- }

Symbolic Names

- mtype = { NUL, REQ, ACK }
  mtype = { ERR, RER }
  mtype = { apple, orange }
- Maximally 255 symbolic names
- mtype m = NUL;
  printf("the message is %e", m)
  prints:
  `the message is NUL`

Two-dimensional arrays

- typedef VECTOR {
  int vector[10]
  }

- VECTOR matrix[5]

- matrix[3].vector[6] = matrix[4].vector[7];

Sparse arrays

- Two-dimensional array produces large state space
- Sparse arrays only store values not equal to 0

Sparse matrix model

```
#define N 4

typedef ENTRY {
  byte row;
  byte col;
  int value
}

ENTRY a[N];

active proctype P() {
  int i = 0;
  int x = 0;
  int r, c;
  a[0].row = 0; a[0].col = 1; a[0].value = -5;
  a[1].row = 0; a[1].col = 3; a[1].value = 8;
  a[2].row = 2; a[2].col = 0; a[2].value = 20;
  a[3].row = 3; a[3].col = 3; a[3].value = -3;
  r = 0;
  do
  :: r > N-1 -> break
  :: else ->
    c = 0;
    do
    :: c > N-1 -> break
    :: else ->
      if /* i == N -> false */
      :: i < N && r == a[i].row && c == a[i].col ->
        x = x + a[i].value;
        i++
      :: else -> true
      fi;
      c++
    od;
    r++
  od;
  assert i == N && x == 20
}
```

Macros

- ***Useful for structuring models***

```
#define N 5

#define ck(ar,x) \
{ \
  d_step { \
    k = 0; \
    do :: k > N-1 -> break \
       :: else -> \
         assert ar[k] >= x; \
         k++ \
    od \
  } \
}
```

```
active proctype P() {
  int a[N];
  int i;
  int k;
  ck(a,0);
  i = 0;
  do :: i > N-1 -> break
     :: else ->
       a[i] = i+1;
       i++
  od;
  ck(a,1)
}
```

Channels

- ***Communication channels*** are used to model distributed systems
- On a channel ***messages*** can be ***sent*** or ***received***
- A channel is declared with initializer specifying the ***channel capacity*** and the **message type**
- chan ch = [capacity] of { typename, ..., typename }
- capacity must not be negative

Send and receive

- ch ! e
  send e on channel ch
- ch ? e
  receive e on channel ch
  (there are some specifics about e *to be discussed later)*
- After ***some process*** has sent a value on ch
  any process may receive that value on ch
  (that includes the sending process!)

Channel example

```
chan request = [0] of { byte };

active proctype Server() {
  byte client = 2;
  byte expected = 2;
  end: /* makes this a valid end state */
  do
  :: request ? client ->
    assert expected == 2
        || 1 - expected == client;
    expected = client
  od
}
```

```
active proctype Client0() {
  request ! 0
}

active proctype Client1() {
  request ! 1
}
```

Channel as parameters

Can you correct the model?

```
chan ch1 = [0] of { byte };

chan ch2 = [0] of { byte, byte };

proctype P(chan c) {
  c!5
}

init {
  run P(ch1);
  run P(ch2)
}
```

Mobility

```
chan c = [1] of { chan };

active proctype P() {
  int x;
  chan m = [0] of { byte };
  c ! m;
  m ? x;
  assert x == 5;
}

active proctype Q() {
  chan m;
  c ? m;
  m ! 5;
}
```

Channels types

- Capacity = 0:
  rendezvous channel

- Capacity > 0:
  buffered channel

Rendezvous Channels

- Channel with capacity = 0
- Consequence:
  the transfer of a message from sender to receiver is ***synchronous*** and executed as a ***single atomic action***

Client-Server Network

```
chan request = [0] of { byte };
chan reply = [0] of { bool };

active proctype Server() {
  byte client;
  end:
  do
  :: request ? client -> reply ! true
  od
}
```

```
active proctype Client0() {
  request ! 0;
  reply ? _
}

active proctype Client1() {
  request ! 1;
  reply ? _
}
```

Not realistic!

Client-Server Network

```
chan request = [0] of { byte };
chan reply = [0] of { byte, byte };

active [2] proctype Server() {
  byte client;
  end:
  do
  :: request ? client -> reply ! client, _pid
  od
}

active [2] proctype Client() {
  byte client, server;
  request ! _pid;
  reply ? client, server;
  assert client == _pid;
}
```

Clients and servers exchange messages

Unfortunately, communication between specific clients and servers is not guaranteed.

Homework

- *Task:*
  Model a client-server network that has peer-to-peer connections.
  Caveat:
  What is transmitted on the peer-to-peer connection should no be visible to clients and servers outside that connection.
  Extension:
  How could a server have connections to two clients? Model a system with 2 servers and 3 clients.

Buffered Channels

- Channel with capacity > 0
- Consequence:
  Messages can be received with a delay

```
chan c = [1] of { byte };
int x;

active proctype P() {
  c ! 1;
  c ! 2-x;
  c ? x;
  assert x == 1;
}

active proctype Q() {
  c ? x
}
```

Channel contents

```
#define N 5

chan c = [1] of { byte };

active proctype P() {
  int k;
  c ! 1;
  do :: c ? k ->
        if :: k > N-1 -> break
           :: else -> c ! k+1
        fi;
     :: else -> break
  od;
  assert k == 5
}
```

Instead of **else** use

- **empty**(c)
- **full**(c)
- **nempty**(c)
- **nfull**(c)

in **do** and **if** statements in alternatives to receive statements.

Channel contents

```
#define N 5

chan c = [1] of { byte };

active proctype P() {
 int k;
 c ! 1;
 do :: c ? k ->
      if :: k > N-1 -> break
         :: else -> c ! k+1
      fi;
    :: empty(c) -> break
 od;
 assert k == 5
}
```

Instead of **else** use

- **empty**(c)
- **full**(c)
- **nempty**(c)
- **nfull**(c)

in **do** and **if** statements in alternatives to receive statements.

Length of channel contents

- Len(c) returns the number of messages in channel c
- Use carefully because partial-order reduction of the SPIN model checker won't work
- We have seen this before: The next-operator of LTL

Random receive

- Should be called differently…
  - It is not random
  - It is not even non-deterministic
- Syntax:
  c ?? e
- Semantics:
  returns the first message in the buffer matching e

Copying and Polling

- A message can be copied from a channel without removing it from the channel:
  ch ? <e> or ch ?? <e>
- Copying has the side effect of (potentially) changing variables
- A message can be polled from a channel without removing it from the channel
  ch ? [const_e] or ch ?? [const_e]
- ***Polling expressions*** can be used in guards unlike ***copying statements***.

Homework

- Model a buffered channel by means of rendezvous channels