Exception Handling in Communicating Sequential Processes

Mads Ohm Larsen

Copenhagen University: Department of Computer Science

Outline

Outline

Motivation

Why Should We Care?

- Reliable software is able to handle exceptions.
- Most programming languages today can handle exceptions internally.
- Using CSP we should be able to let other processes handle an exception.

Outline

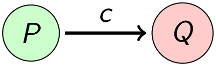
Back to Basics

What is Communication?

- A communication is an event done by two or more processes in parallel.

One-to-one

$$P = c!x \rightarrow P'$$
$$Q = c?x \rightarrow Q'(x)$$
$$O_2O = P \,||\, Q$$



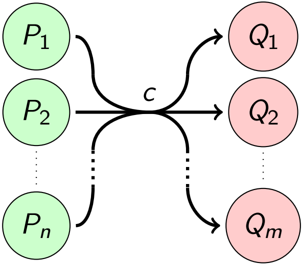
Back to Basics

What is Communication?

- Any-to-any channels can be "created" with the use of the interleaving operator.

Any-to-any

$$P_i = c!x \rightarrow P_i'$$
$$Q_j = c?x \rightarrow Q_j'(x)$$
$$A_2A = \left(\mathop{|||}_{i \in 1..n} P_i \right) \| \left(\mathop{|||}_{j \in 1..m} Q_j \right)$$

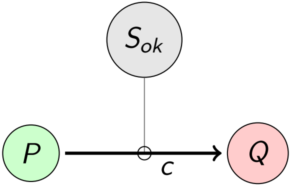


Outline

Supervisor Paradigm

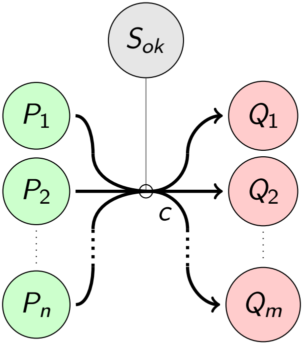
Meet the Supervisor

- A supervisor overlooks the channel.
- It controls which communication events are allowed, by engaging in them.



Supervisor Paradigm

Meet the Supervisor



Supervisor Paradigm

Meet the Supervisor

- Let us look at the supervisor process.

Supervisor

$$S_{ok} = \left(d : \{c.m \mid m \in \alpha c\}\right) \rightarrow S_{ok}$$

- Right now this allows for all communication, when run in parallel, however it can be modified for both poison, retirement and exception handling.

Poison

Killing a Network

- Each process should be able to shut down.
- In various implementations of CSP we have a poison construct to shut down a network.
- The supervisor process can be altered to encompass poison.
- It must have a unique event, for each other process, that should be able to poison the channel, it overlooks.

Poison

Killing a Network

Poison

$$S_{ok} = \left(\left(d : \{c.m \mid m \in \alpha c\}\right) \rightarrow S_{ok}\right) \,\square\, \left(\mathop{\square}_{id} c_{p_{id}} \rightarrow S_e\right)$$

$$S_e = c_{poison} \rightarrow S_e \,\square\, SKIP$$

$$P_i = \left(c!x \rightarrow P_i'\right) \,\square\, \left(c_{poison} \rightarrow P_{p_i}\right)$$

$$Q_j = \left(c?x \rightarrow Q_j'(x)\right) \,\square\, \left(c_{poison} \rightarrow Q_{p_j}\right)$$

Poison

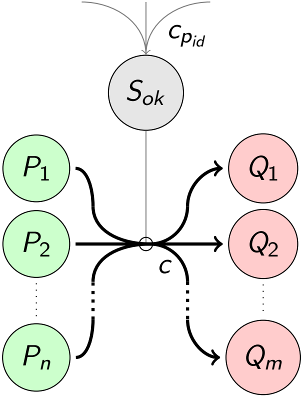
Killing a Network

Poison

$$POISON_{A_2A} = \left(\underset{i \in 1..n}{|||} P_i \right) \,||\, \left(\underset{j \in 1..m}{|||} Q_j \right) \,||\, S_{ok}$$

Poison

Killing a Network



Retirement

Shutting Down a Network

- Retirement is poisons less aggressive brother.
- We count reader and writers. A channel is retired if either reaches zero.

Retirement

Shutting Down a Network

Retirements Supervisor

$$
\begin{aligned}
S_{ok}(0, \_) &= S_e \\
S_{ok}(\_, 0) &= S_e \\
S_{ok}(n, m) &= \Big((d \ : \ \{c.me \mid me \in \alpha c\}) \rightarrow S_{ok}\,(n, m) \Big) \\
&\quad \mathop{\square}_{id} \Big(c_{rw_{id}} \rightarrow S_{ok}(n-1, m) \Big) \\
&\quad \mathop{\square}_{id} \Big(c_{rr_{id}} \rightarrow S_{ok}(n, m-1) \Big)
\end{aligned}
$$

$$S_e = c_{retire} \rightarrow S_e \ \square \ SKIP$$

Retirement

Shutting Down a Network

Retirement Network

$$RETIRE_{A_2A} = \left(\mathop{|||}_{i \in 1..n} P_i \right) \| \left(\mathop{|||}_{j \in 1..m} Q_j \right) \| S_{ok}(n, m)$$

Outline

Exception Handling

How Do We Handle Exceptions?

- CSP already offers to interrupt a process via the interrupt operator.

Interrupt

$$P \,\Delta\, Q$$

- This behaves as P but is interrupted on the first occurrence of an event of Q.

Exception Handling

How Do We Handle Exceptions?

- We call an outside-error a catastrophe $\lightning$.
- A process that behaves as P up until a catastrophe and then behaves as Q is defined by

Catastrophe

$$P \mathbin{\hat{\lightning}} Q = P \;\Delta\; (\lightning \to Q)$$

- Roscoe continues this, and creates the throw operator

Throw operator

$$P \;\Theta_{x:A}\; Q(x)$$

Exception Handling

How Do We Handle Exceptions?

- We can catch all errors in a process with this throw operator.

Caught

$$P_i = (c!x \rightarrow P_i') \; \Theta_{error} \; P_{e_i}$$
$$Q_j = (c?x \rightarrow Q_j'(x)) \; \Theta_{error} \; Q_{e_j}$$

- The P_{e_i} and Q_{e_j} processes could be telling the supervisor that the process in hand is in an exception state.

Handled

$$P_{e_i} = c_{e_i} \rightarrow SKIP$$
$$Q_{e_j} = c_{e_j} \rightarrow SKIP$$

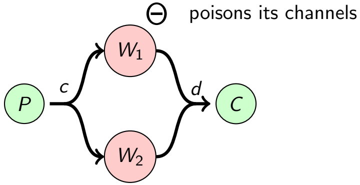
Fail-stop

Press the Big Red Button

- Fail-stop is just like poison.
- It occurs when a process goes into an exception state.

Fail-stop

Press the Big Red Button



Fail-stop

Press the Big Red Button

```
from pycsp_import import *

@process
def producer(job_out):
  for i in range(-10, 11):
    job_out(i)

@process(fail_type = FAILSTOP)
def worker(job_in, job_out):
  while True:
    x = job_in()
    job_out(1.0/x)

@process
def consumer(job_in):
  try:
    while True:
      x = job_in()
      print x
  except ChannelFailstopException:
    print "Caught the exception"

c = Channel()
d = Channel()
```

```
Parallel(
  producer(-c),
  3 * worker(+c, -d),
  consumer(+d)
)
```

```
-0.1
-0.111111111111
-0.125
-0.142857142857
-0.166666666667
-0.2
-0.25
-0.333333333333
-0.5
-1.0
1.0
Caught the exception
```

Retire-like Fail-stop

Press the Slightly Smaller Red Button

- Of course, retire-like fail-stop works like retire.

Retire-like network

$$P_0 = P'_0 = SKIP$$
$$P_x = c!x \rightarrow P_{x-1} \ominus P'_x$$
$$P'_x = d!x \rightarrow P'_{x-1}$$
$$F = c?x \rightarrow f!(x \cdot 2) \rightarrow F$$
$$W = d?x \rightarrow f!(x \cdot 2) \rightarrow W$$
$$C = f?x \rightarrow print!x \rightarrow C$$

$$Rnet = \Big(I(P_{10}) \parallel \big(I(F) \mid\mid\mid I(W) \big) \parallel I(C) \Big) \parallel S_{ok}(1,1) \parallel T_{ok}(1,1) \parallel U_{ok}(2,1)$$

Retire-like Fail-stop

Press the Slightly Smaller Red Button

```
from pycsp_import import *

@process(fail_type = RETIRELIKE)
def producer(cout, dout, job_start,
             job_end):
    try:
      for i in range(job_start, job_end):
        cout(i)
    except ChannelRetireLike...
           FailstopException:
      for i in range(i, job_end):
        dout(i)

@process(fail_type = RETIRELIKE)
def failer(cin, fout):
  while True:
    x = cin()
    fout(x*2)
    raise Exception("failed hardware")

@process(fail_type = RETIRELIKE)
def worker(din, fout):
  while True:
    x = din()
    fout(x*2)
```

```
@process(fail_type = RETIRELIKE)
def consumer(finish):
  while True:
    x = finish()
    print x

c = Channel()
d = Channel()
f = Channel()

Parallel(
  producer(-c, -d, -10, 10),
  failer(+c, -f),
  worker(+d, -f),
  consumer(+f)
)
```

Retire-like Fail-stop

Press the Slightly Smaller Red Button

```
-20
failed hardware
-18
-16
-14
-12
-10
-8
-6
-4
-2
0
2
4
6
8
10
12
14
16
18
```

Outline

Checkpointing

We Can Roll Back Our Mistakes

- We want a way to roll back to last valid checkpoint.
- A checkpoint is rendered invalid on side-effects, from the process, that is, printing, communicating, writing to files and so on.

Checkpointing

We Can Roll Back Our Mistakes

- Let us create a process $Ch(P)$ which checkpoints P.
- As we want to keep the latest checkpoint, we need an auxiliary process $Ch2(P, Q)$.
- Here P is the process and Q is the latest checkpoint.

Checkpointing Process

$$Ch(P) = Ch2(P, P)$$

Checkpointing

We Can Roll Back Our Mistakes

Checkpointing Process

$$Ch(P) = Ch2(P, P)$$

- If Ⓒ is a checkpoint event, Ⓡ is a roll back event, and $P = \big(x : A \rightarrow P(x)\big)$ then $Ch2(P, Q)$ can be defined as

Aux. Checkpointing

$$\begin{aligned} Ch2(P, Q) = \Big(& x : A \rightarrow Ch2(P(x), Q) \\ & \mid \text{Ⓒ} \rightarrow Ch2(P, P)\Big) \ominus \text{Ⓡ} \rightarrow Ch2(Q, Q) \end{aligned}$$

Checkpointing

We Can Roll Back Our Mistakes

- With this we can checkpoint an entire network with

Checkpoint a Network

$$Ch(P \,||\, Q)$$

- ... or individual processes with

Checkpoint a Process

$$Ch(P) \,||\, Ch(Q)$$

Checkpointing

We Can Roll Back Our Mistakes

- Having just one ⓒ will require every process to checkpoint at the same time.
- A better way is to have all processes which engages in a communication to checkpoint at the same time.
- Recalling that processes on each side of the communication are interleaving, only two of them will checkpoint, the sender and the receiver.

Checkpointing

We Can Roll Back Our Mistakes

- This requires a small change to $Ch2$.

New Aux. Checkpointing

$$
\begin{aligned}
Ch2(P, Q) = \Big(& x : A \rightarrow Ch2(P(x), Q) \\
& \mathop{\Box}_{c \in \alpha P} (ⓒ_c \rightarrow Ch2(P, P)) \Big) \ominus \\
& \mathop{\Box}_{c \in \alpha P} ⓡ_c \rightarrow Ch2(Q, Q)
\end{aligned}
$$

Checkpointing

We Can Roll Back Our Mistakes

- The supervisor will have to be in on the checkpointing, so we change it to

New Aux. Checkpointing

$$S_{ok} = \left(d \;:\; \{c.me \mid me \in c\}\right) \rightarrow ⓒ_c \rightarrow S_{ok}$$
$$\square \left(ⓡ_c \rightarrow S_{ok}\right)$$

- To keep it simple this is missing all the poison and retire abilities.

Checkpointing

We Can Roll Back Our Mistakes

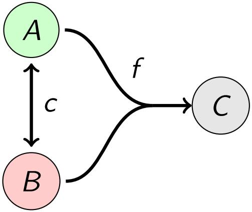


Figure: Programming model

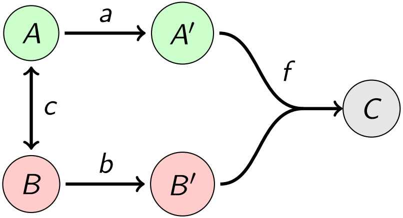


Figure: CSP model

Checkpointing

We Can Roll Back Our Mistakes

Checkpointing network

$$A = c!(\text{"}Ping\text{"}) \rightarrow c?y \rightarrow a!y \rightarrow A$$
$$A' = a?x \rightarrow f!x \rightarrow A'$$
$$B = c?x \rightarrow c!(\text{"}Pong\text{"}) \rightarrow b!x \rightarrow B$$
$$B' = b?x \rightarrow f!x \rightarrow B'$$
$$C_0 = f_{poison} \rightarrow SKIP$$
$$C_n = f?x \rightarrow print!x \rightarrow C_{n-1}$$

$$CPNet = \Big(Ch(A) \,||\, Ch(B)\Big) \,||\, \Big(Ch(A') \,|||\, Ch(B')\Big) \,||\, Ch(C_{100})$$
$$||\, S_{ok}(2,2) \,||\, T_{ok}(1,1) \,||\, U_{ok}(1,1) \,||\, V_{ok}(2,1)$$

Checkpointing

We Can Roll Back Our Mistakes

```
from pycsp_import import *
from random import randint

@process(fail_type = CHECKPOINT)
def A(cout, cin, fout):
  while True:
    cout("Ping")
    fout(cin())

@process(fail_type = CHECKPOINT,
         retries = -1)
def B(cout, cin, fout):
  while True:
    x = cin()
    cout("Pong")
    # This next line fails
    # roughly half the time
    1/randint(0, 1)
    fout(x)

@process(fail_type = CHECKPOINT)
def C(fin, num):
  i = load(i = 1)
  for i in range(i, num):
    f = fin()
    print i, f
  poison(fin)
```

```
c = Channel()
f = Channel()

Parallel(
  A(-c, +c, -f),
  B(-c, +c, -f),
  C(+f, 100)
)
```

```
0 Ping
1 Pong
2 Ping
3 Pong
4 Ping
5 Pong
6 Ping
7 Pong
8 Ping
...





...
99 Pong
```

Outline

Conclusions

- Presented a supervisor paradigm
  - This is helping poison, retirement as well as exception handling.
- Shown and implemented fail-stop and retire-like fail-stop.
- Shown and implemented checkpointing and roll back.

Outline

Future Work

- Only works on on-processes, as described by Roscoe in *On the expressiveness of CSP, feb. 2011*
- If the process is not on the form $P = (x : A \rightarrow P(x))$ we cannot create $Ch2(P, Q)$.
- Let us say we have two processes P and Q

"On"-process

$$P = c \rightarrow \left(a \rightarrow STOP \ \sqcap \ b \rightarrow STOP\right)$$

$$Q = c \rightarrow a \rightarrow STOP \ \sqcap \ c \rightarrow b \rightarrow STOP$$

- These are equivalent, however, they are checkpointed in different ways after c.

Future Work

“On”-process checkpoint

$$P \Rightarrow Ch2(a \rightarrow STOP \sqcap b \rightarrow STOP,$$
$$a \rightarrow STOP \sqcap b \rightarrow STOP)$$

and

$$Q \Rightarrow Ch2(a \rightarrow STOP, a \rightarrow STOP)$$
$$\text{or} \quad Ch2(b \rightarrow STOP, b \rightarrow STOP)$$

- Some investigation needs to be put into whether or not it is possible to create $Ch2(P, Q)$ for all processes.

Future Work

- The programmer needs to make sure that the processes do not have side-effects. No warnings are given.

Future Work

- The checkpoints could be used as a starting point for other processes.
  - In a real-world application, the processes could be stopped, moved and restarted at the same point on different hardware.

Replayable Messages

- We want to be able to replay messages sent to a process.
- If a process goes into an exception state, an intermediate process should replay all still valid messages to the same channel.
    - Of course only applicable on one-to-any and any-to-any channels.

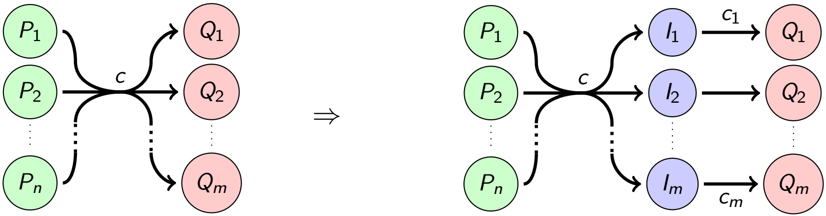
Replayable Messages

- A message is valid, as long as the process receiving it says it is valid.
    - That is, a process receiving can deem a message invalid.
- When deeming any one message invalid, you deem all prior messages invalid as well.

Replayable Messages

- The intermediate process has a list of messages.
- It can add to this list as well as delete the list entirely.
- Of course it is able to replay all messages, removing them individually from the list as well.

Replayable Messages



Replayable Messages

Intermediate Process

$$I_j = R_{()}$$

where

$$R_s = c?x \rightarrow c_j!x \rightarrow R_{s \frown \{x\}} \; \square \; c_j.replay \rightarrow R'_s$$
$$\square \; c_j.delete \rightarrow R_{()}$$
$$R'_{()} = R_{()}$$
$$R'_{\{x\} \frown s} = c!x \rightarrow R'_s$$

Replayable Messages

```
from pycsp_import import *

@process
def producer(job_out):
  for i in range(-10, 0):
    job_out(i)

  job_out("replay")

  for i in range(0, 11):
    job_out(i)

  while True:
    job_out("retire")

@process
def worker(job_in, job_out):
  while True:
    x = job_in()
    job_out(x * 2)
```

```
@process
def replayer(job_in, job_out, replay):
  jobs = []
  while True:
    x = job_in()

    if x == "delete":
      jobs = []
    elif x == "replay":
      for j in jobs:
        replay(j)
      jobs = []
    elif x == "retire":
      raise ChannelRetireException
    else:
      jobs.append(x)
      job_out(x)

@process
def consumer(job_in):
  while True:
    print job_in()
```

Replayable Messages

```
c = Channel()
c1,c2,c3 = Channel(),Channel(),Channel()
d = Channel()

Parallel(
  producer(-c),
  replayer(+c, -c1, -c),
  replayer(+c, -c2, -c),
  replayer(+c, -c3, -c),
  worker(+c1, -d),
  worker(+c2, -d),
  worker(+c3, -d),
  consumer(+d)
)
```

```
-20
-18
-16
-14
-12
-10
-8
-6
-4
-2
>>> -18
0
>>> -12
2
>>> -6
4
6
8
10
12
14
16
18
20
```

Thank you very much

Questions?