Exception Handling in Communicating Sequential Processes

Mads Ohm Larsen
Copenhagen University: Department of Computer Science
Outline

1 Motivation

2 Back to Basics

3 Supervisor Paradigm
   Poison
   Retirement

4 Exception Handling
   Fail-stop
   Retire-like Fail-stop

5 Checkpointing

6 Conclusions

7 Future Work
Outline

1 Motivation

2 Back to Basics

3 Supervisor Paradigm
   Poison
   Retirement

4 Exception Handling
   Fail-stop
   Retire-like Fail-stop

5 Checkpointing

6 Conclusions

7 Future Work
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

1 Motivation
2 Back to Basics
3 Supervisor Paradigm
   Poison
   Retirement
4 Exception Handling
   Fail-stop
   Retire-like Fail-stop
5 Checkpointing
6 Conclusions
7 Future Work
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

\[ P_i = c!x \rightarrow P'_i \]
\[ Q_j = c?x \rightarrow Q'_j(x) \]

\[ A_2A = \left( \parallel_{i \in 1..n} P_i \right) \parallel \left( \parallel_{j \in 1..m} Q_j \right) \]
Outline

1 Motivation
2 Back to Basics
3 Supervisor Paradigm
   Poison
   Retirement
4 Exception Handling
   Fail-stop
   Retire-like Fail-stop
5 Checkpointing
6 Conclusions
7 Future Work
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.

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

- 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

\[
S_{ok} = \left( (d : \{c.m \mid m \in \alpha c\}) \Rightarrow S_{ok} \right) \square \left( \square_{id} c_{pid} \Rightarrow S_e \right)
\]

\[
S_e = c_{poison} \Rightarrow S_e \square \text{SKIP}
\]

\[
P_i = (c!x \Rightarrow P'_i) \square (c_{poison} \Rightarrow P_{p_i})
\]

\[
Q_j = (c?x \Rightarrow Q'_j(x)) \square (c_{poison} \Rightarrow Q_{p_j})
\]
Poison
Killing a Network

Poison

\[ \text{POISON}_{A2A} = \left( \big|\big| \big| P_i \big|\big| \right) \parallel \left( \big|\big| \big| Q_j \big|\big| \right) \parallel S_{\text{ok}} \]
Poison
Killing a Network

\[ S_{ok} \]

\[ S_{ok} \]

\[ c_{pid} \]

\[ P_1 \]
\[ P_2 \]
\[ P_n \]

\[ Q_1 \]
\[ Q_2 \]
\[ Q_m \]
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

\[ S_{ok}(0, -) = S_e \]
\[ S_{ok}(-, 0) = S_e \]
\[ S_{ok}(n, m) = ( (d : \{ c.me \mid me \in \alpha c \}) \rightarrow S_{ok}(n, m)) \]
\[ \square_{id}(c_{rwid} \rightarrow S_{ok}(n - 1, m)) \]
\[ \square_{id}(c_{rrid} \rightarrow S_{ok}(n, m - 1)) \]

\[ S_e = c_{retire} \rightarrow S_e \square SKIP \]
Retirement
Shutting Down a Network

\[
RETIRE_{A2A} = \left( \big||\big| P_i \right| \big) \big|\big| \left( \big||\big| Q_j \right| \big) \big| S_{ok}(n, m) \right.
\]
Outline

1 Motivation
2 Back to Basics
3 Supervisor Paradigm
   Poison
   Retirement
4 Exception Handling
   Fail-stop
   Retire-like Fail-stop
5 Checkpointing
6 Conclusions
7 Future Work
Exception Handling
How Do We Handle Exceptions?

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

\[ P \triangle Q \]

- 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 $\n$.
- A process that behaves as $P$ up until a catastrophe and then behaves as $Q$ is defined by

$$P \n Q = P \bigtriangleup (\n \rightarrow Q)$$

- Roscoe continues this, and creates the 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

\[
P_i = (c!x \rightarrow P'_i) \Theta_{\text{error}} P_{e_i}
\]
\[
Q_j = (c?x \rightarrow Q'_j(x)) \Theta_{\text{error}} Q_{e_j}
\]

- 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

\[
P_{e_i} = c_{e_i} \rightarrow \text{SKIP}
\]
\[
Q_{e_j} = c_{e_j} \rightarrow \text{SKIP}
\]
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

\[ \Theta \] poisons its channels
Fail-stop
Press the Big Red Button

```python
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**

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

\[
Rnet = \left( I(P_{10}) \parallel (I(F) \parallel I(W)) \parallel I(C) \right)
\]
\[
\parallel S_{ok}(1, 1) \parallel T_{ok}(1, 1) \parallel U_{ok}(2, 1)
\]
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
20
21
Outline

1 Motivation
2 Back to Basics
3 Supervisor Paradigm
   Poison
   Retirement
4 Exception Handling
   Fail-stop
   Retire-like Fail-stop
5 Checkpointing
6 Conclusions
7 Future Work
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
  \[ P = (x : A \rightarrow P(x)) \]
  then \( Ch2(P, Q) \) can be defined as

Aux. CheckPointing

\[ Ch2(P, Q) = \left( x : A \rightarrow Ch2(P(x), Q) \right) \]

\[ \mid ⬀ \rightarrow Ch2(P, P) \Theta ⬤ \rightarrow Ch2(Q, Q) \]
Checkpointing
We Can Roll Back Our Mistakes

- With this we can checkpoint an **entire network** with

  $Ch(P \parallel Q)$

- ... or **individual processes** with

  $Ch(P) \parallel Ch(Q)$
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.
Checkpoiniting
We Can Roll Back Our Mistakes

- This requires a small change to $Ch2$.

New Aux. Checkpointing

\[
Ch2(P, Q) = \left( x : A \rightarrow Ch2(P(x), Q) \right.
\]

\[
\begin{align*}
\square_{c \in \alpha P} \left( (\circlearrowleft_{c} \rightarrow Ch2(P, P)) \right) \Theta \\
\square_{c \in \alpha P} (\circlearrowright_{c} \rightarrow Ch2(Q, Q))
\end{align*}
\]
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**

\[
S_{ok} = \left( d : \{ c.me \mid me \in c \} \right) \rightarrow \bigcirc_c \rightarrow S_{ok}
\]

\[
\square \left( \overline{r}_c \rightarrow S_{ok} \right)
\]

- 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

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

\[ CPNet = \left( Ch(A) \parallel Ch(B) \right) \parallel \left( Ch(A') \parallel\parallel Ch(B') \right) \parallel Ch(C_{100}) \parallel S_{ok}(2, 2) \parallel T_{ok}(1, 1) \parallel U_{ok}(1, 1) \parallel V_{ok}(2, 1) \]
Checkpointing

We Can Roll Back Our Mistakes

```python
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

1 Motivation

2 Back to Basics

3 Supervisor Paradigm
   Poison
   Retirement

4 Exception Handling
   Fail-stop
   Retire-like Fail-stop

5 Checkpointing

6 Conclusions

7 Future Work
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

1 Motivation

2 Back to Basics

3 Supervisor Paradigm
   Poison
   Retirement

4 Exception Handling
   Fail-stop
   Retire-like Fail-stop

5 Checkpointing

6 Conclusions

7 Future Work
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**

\[
\begin{align*}
P &= c \rightarrow (a \rightarrow STOP \sqcap b \rightarrow STOP) \\
Q &= c \rightarrow a \rightarrow STOP \sqcap c \rightarrow b \rightarrow STOP
\end{align*}
\]

- These are equivalent, however, they are checkpointed in different ways after \( c \).
Future Work

“On”-process checkpoint

\[ P \Rightarrow Ch2(a \rightarrow STOP \sqcap b \rightarrow STOP, \]
\[ \quad a \rightarrow STOP \sqcap b \rightarrow STOP) \]

and

\[ Q \Rightarrow Ch2(a \rightarrow STOP, a \rightarrow STOP) \]
\[ \quad or \quad Ch2(b \rightarrow STOP, b \rightarrow STOP) \]

- 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

\[ P_1 \xrightarrow{c} Q_1 \xrightarrow{c} P_2 \xrightarrow{c} Q_2 \xrightarrow{c} P_n \xrightarrow{c} Q_m \implies P_1 \xrightarrow{c} I_1 \xrightarrow{c} Q_1 \xrightarrow{c} I_2 \xrightarrow{c} Q_2 \xrightarrow{c} I_m \xrightarrow{c} Q_m \]
Replayable Messages

Intermediate Process

\[ I_j = R() \]

where

\[ R_s = c?x \rightarrow c_j!x \rightarrow R_s\sim\{x\} \]

\[ \square c_j.replay \rightarrow R'_s \]

\[ \square c_j.delete \rightarrow R() \]

\[ R'_() = R() \]

\[ R'\{x\}_s \sim = c!x \rightarrow R'_s \]
Replayable Messages

```python
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()
```

Mads Ohm Larsen — Exception Handling in CSP — 4. sep. 2012
Slide 51/53
Replayable Messages

```plaintext
1  c = Channel()
2  c1, c2, c3 = Channel(), Channel(), Channel()
3  d = Channel()
4
5  Parallel(
6      producer(-c),
7      replayer(+c, -c1, -c),
8      replayer(+c, -c2, -c),
9      replayer(+c, -c3, -c),
10     worker(+c1, -d),
11     worker(+c2, -d),
12     worker(+c3, -d),
13     consumer(+d)
14  )
```
Thank you very much

Questions?