Exception Handling and Checkpointing in CSP

Mads Ohm Larsen
Copenhagen University: Department of Computer Science
Outline

1 Motivation
Outline

1 Motivation
2 Back to Basics
Outline

1 Motivation

2 Back to Basics

3 Supervisor Paradigm
   Poison
   Retirement
Outline

1 Motivation
2 Back to Basics
3 Supervisor Paradigm
   Poison
   Retirement
4 Exception Handling
   Fail-stop
   Retire-like Fail-stop
Outline

1. Motivation
2. Back to Basics
3. Supervisor Paradigm
   - Poison
   - Retirement
4. Exception Handling
   - Fail-stop
   - Retire-like Fail-stop
5. Checkpointing
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
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 Work Left to be Done
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 Work Left to be Done
Motivation
Why Should We Care?

- Reliable software is able to handle exceptions.
Motivation
Why Should We Care?

- Reliable software is able to handle exceptions.
- Most programming languages today can handle exceptions.
Motivation
Why Should We Care?

- Reliable software is able to handle exceptions.
- Most programming languages today can handle exceptions internally.
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 Work Left to be Done
Back to Basics

What is Communication?

- A communication is an event done by two or more processes in parallel.
Back to Basics
What is Communication?

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

One-to-one

\[ P = c!x \rightarrow P' \]
\[ Q = c?x \rightarrow Q'(x) \]
\[ O_2 O = P \parallel Q \]
Back to Basics
What is Communication?

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

### One-to-one

\[
P = c!x \rightarrow P' \\
Q = c?x \rightarrow Q'(x) \\
O_2O = P \parallel Q
\]
Back to Basics

What is Communication?

- Any-to-any channels can be “created” with the use of the interleaving operator.
Back to Basics
What is Communication?

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

Any-to-any

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

\[
A_2A = \left( \big|| \big|| P_i \big) \big|| \left( \big|| \big|| Q_j \big) \right)
\]

\[
i \in 1..n \\
j \in 1..m
\]
Back to Basics
What is Communication?

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

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

\[ A_2A = \left( \bigl\| \bigl\| P_i \bigr\| \right) \| \left( \bigl\| \bigl\| Q_j \bigr\| \right) \]
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 Work Left to be Done
Supervisor Paradigm
Meet the Supervisor

- A supervisor overlooks the channel.
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

- 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 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.
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.
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.
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.
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

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

\[
S_e = c_{poison} \rightarrow S_e \square SKIP
\]

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

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

\[
\text{POISON}_{A_2A} = \left( \bigparallel_{i \in 1..n} P_i \right) \ || \ \left( \bigparallel_{j \in 1..m} Q_j \right) \ || \ S_{ok}
\]
Poison

Killing a Network

\[ S_{ok} \]

\[ c_{pid} \]

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

\[ Q_1 \]
\[ Q_2 \]
\[ Q_m \]
Retirement
Shutting Down a Network

- Retirement is poisons less aggressive brother.
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

\[
S_{ok}(0, -) = S_e \\
S_{ok}(-, 0) = S_e \\
S_{ok}(n, m) = \left( (d : \{ c.me \mid me \in \alpha c \}) \rightarrow S_{ok}(n, m) \right) \\
\square(id \ (c_{rw_id} \rightarrow S_{ok}(n - 1, m)) \\
\square(id \ (c_{rr_id} \rightarrow S_{ok}(n, m - 1)))
\]

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

Retirement Network

\[ RETIRE_{A2A} = \left( \bigcirclearrowleft_{i \in 1..n} P_i \right) \parallel \left( \bigcirclearrowleft_{j \in 1..m} Q_j \right) \parallel S_{ok}(n, m) \]
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 Work Left to be Done
Exception Handling
How Do We Handle Exceptions?

• CSP already offers to interrupt a process via the interrupt operator.
Exception Handling
How Do We Handle Exceptions?

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

\[ P \Delta Q \]
Exception Handling

How Do We Handle Exceptions?

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

\[
P \Delta Q
\]

- 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 $\nabla$.
Exception Handling
How Do We Handle Exceptions?

- We call an outside-error a catastrophe $\not\circ$.
- A process that behaves as $P$ up until a catastrophe and then behaves as $Q$ is defined by

$$Catastrophe P \not\circ \rightarrow Q = P \triangle (\not\circ \rightarrow Q)$$

Roscoe continues this, and creates the throw operator

Throw operator $P \Theta x$:

$A Q (x)$
Exception Handling
How Do We Handle Exceptions?

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

\[
P \overset{\#}{\Rightarrow} Q = P \Delta (\# \rightarrow Q)
\]
Exception Handling
How Do We Handle Exceptions?

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

$$P \xrightarrow{\downarrow} Q = P \Delta (\downarrow \rightarrow Q)$$

- Roscoe continues this, and creates the throw operator
Exception Handling

How Do We Handle Exceptions?

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

\[
P \overset{\triangledown}{\rightarrow} Q = P \Delta (\overset{\triangledown}{\rightarrow} Q)
\]

• 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.
Exception Handling

How Do We Handle Exceptions?

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

Caught

\[
P_i = (c!x \rightarrow P_i') \Theta_{error} P_{e_i}
\]

\[
Q_j = (c?x \rightarrow Q_j'(x)) \Theta_{error} Q_{e_j}
\]
Exception Handling
How Do We Handle Exceptions?

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

Caught

\[
P_i = (c!x \to P_i') \Theta_{error} P_{e_i}
\]
\[
Q_j = (c?x \to Q_j')(x) \Theta_{error} Q_{e_j}
\]

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

\[ \Theta \text{ poisons its channels} \]

\[ P \xrightarrow{c} W_1 \xrightarrow{d} C \]

\[ W_1 \xrightarrow{c} P \]

\[ W_2 \xrightarrow{d} C \]
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"

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 Fail-stop

Press the Slightly Smaller Red Button

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

**Retire-like network**

\[
P_0 = P'_0 = \text{SKIP}
\]

\[
P_x = c!x \rightarrow P_{x-1} \odot P'_x
\]

\[
P'_x = d!x \rightarrow P'_{x-1}
\]

\[
F = c?x \rightarrow f!(x \cdot 2) \rightarrow F
\]

\[
W = d?x \rightarrow f!(x \cdot 2) \rightarrow W
\]

\[
C = f?x \rightarrow \text{print!}x \rightarrow C
\]

\[
R_{net} = \left( I(P_{10}) \parallel (I(F) \parallel I(W)) \parallel I(C) \right)
\]

\[
\parallel S_{ok}(1, 1) \parallel T_{ok}(1, 1) \parallel U_{ok}(2, 1)
\]
Retire-like Fail-stop
Press the Slightly Smaller Red Button

```python
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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>failed hardware</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>17</td>
<td>10</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
</tr>
<tr>
<td>19</td>
<td>14</td>
</tr>
<tr>
<td>20</td>
<td>16</td>
</tr>
<tr>
<td>21</td>
<td>18</td>
</tr>
</tbody>
</table>
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 Work Left to be Done
Checkpointing
We Can Roll Back Our Mistakes

- We want a way to roll back to last valid checkpoint.
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.
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 $P$. 
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 $Ch2(P, Q)$. 
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
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

\[ Ch(P) = Ch2(P, P) \]
Checkpointing
We Can Roll Back Our Mistakes

Checkpointing Process

\[ Ch(P) = Ch2(P, P) \]

- If \( \mathcal{C} \) is a checkpoint event, \( \mathcal{R} \) is a roll back event, and 
  \[ P = (x : A \rightarrow P(x)) \] 
  then \( Ch2(P, Q) \) can be defined as
Checkpointing
We Can Roll Back Our Mistakes

Checkpointing Process

\[ Ch(P) = Ch2(P, P) \]

- If \( \odot \) is a checkpoint event, \( \overline{\circ} \) is a roll back event, and \( P = (x : A \rightarrow P(x)) \) then \( Ch2(P, Q) \) can be defined as

Aux. Checkpointing

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

\[ | \odot \rightarrow Ch2(P, P) \Theta \overline{\circ} \rightarrow Ch2(Q, Q) \]
Checkpointing
We Can Roll Back Our Mistakes

• With this we can checkpoint an entire network with
Checkpointing
We Can Roll Back Our Mistakes

- With this we can checkpoint an entire network with $Ch(P || Q)$
Checkpointing
We Can Roll Back Our Mistakes

- With this we can checkpoint an entire network with
  \[ Ch(P \parallel Q) \]

- ... or individual processes with
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.
Checkpointing
We Can Roll Back Our Mistakes

- Having just one $\mathbb{C}$ 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.
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 $Ch2$. 
Checkpointing
We Can Roll Back Our Mistakes

- This requires a small change to \( Ch2 \).

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

\[
\square_{c \in \alpha P} (\mathcal{C}_c \rightarrow Ch2(P, P))
\]

\[
\square_{c \in \alpha P} (\mathcal{O}_c \rightarrow Ch2(Q, Q))
\]
Checkpointing
We Can Roll Back Our Mistakes

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

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

$$\Box \left( \overline{r}_c \rightarrow S_{ok} \right)$$

- To keep it simple this is missing all the poison and retire abilities.
Checkpointing
We Can Roll Back Our Mistakes

Checkpointing network

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

\[ CPNet = \left( Ch(A) \parallel Ch(B) \right) \parallel \left( Ch(A') \parallel Ch(B') \right) \parallel Ch(C_{100}) \parallel S_{\text{ok}}(2, 2) \parallel T_{\text{ok}}(1, 1) \parallel U_{\text{ok}}(1, 1) \parallel V_{\text{ok}}(2, 1) \]
Checkpointing
We Can Roll Back Our Mistakes

Figure: Programming model

Figure: CSP model
Checkpointing

We Can Roll Back Our Mistakes

```python
from pycsp 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)
```

```python
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
...
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 Work Left to be Done
Conclusions

- Presented a **supervisor paradigm**
  - This is helping **poison**, **retirement** as well as **exception handling**.
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.
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 Work Left to be Done
Work Left to be Done

• Only works on on-processes, as described by Roscoe in *On the expressiveness of CSP, feb. 2011*
Work Left to be Done

- 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 $Ch2(P, Q)$. 
Work Left to be Done

- 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

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

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

- These are equivalent, however, they are checkpointed in different ways after $c$. 
Work Left to be Done

“On”-process checkpoint

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

and

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

or

\[
Ch2(b \rightarrow STOP, b \rightarrow STOP)
\]

• Some investigation needs to be put into whether or not it is possible to create \(Ch2(P, Q)\) for all processes.
Work Left to be Done

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

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

Questions?