Communicating Process Architectures 2012
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2012

Exception Handling and Checkpointing in CSP

Mads Ohm LARSEN [a,1] and Brian VINTER [b]

[a] *Department of Computer Science, University of Copenhagen*
[b] *Niels Bohr Institute, University of Copenhagen*

Abstract. This paper describes work in progress. It presents a new way of looking at some of the basics of CSP. The primary contributions are exception handling and checkpointing of processes and the ability to roll back to a known checkpoint. Channels are discussed as communication events which are monitored by a supervisor process. The supervisor process is also used to formalise poison and retire events. Exception handling and checkpointing are used as means of recovering from an error. The supervisor process is central to checkpointing and recovery as well. Three different kinds of exception handling are discussed: fail-stop, retire-like fail-stop, and checkpointing. Fail-stop works like poison, and retire-like fail-stop works like retire. Checkpointing works by telling the supervisor process to roll back both participants in a communication event, to a state immediately after their last successful communication. Only fail-stop exceptions have been implemented in PyCSP at this point.

Keywords. CSP, PyCSP, Exceptions, Checkpoints, Algebra, Channels

Introduction

Exceptions can occur in any type of software, however reliable software should be able to handle these exceptions. Currently CSP offers interrupts [1] and has a throw operator [2] to handle exceptions. These exceptions are internal, however other processes in a network might want to know about them. In this paper we want to propagate exceptions throughout a network. These exceptions would trigger a checkpointing mechanism, which would roll back a pair of processes to a know working state.

To get an understanding of the inner workings of CSP, the basics of channels, poison and retire will be discussed in sections 1, 2 and 3 respectively. Together with poison a supervisor paradigm will be developed. This supervisor is critical for telling other processes how to poison a network, but will also be useful for telling other processes about exceptions. Section 4 contains a discussion on how to handle exceptions using CSP and leads up to the reasoning behind and discussion of checkpointing in section 4.4.4.

This is work in progress and a working implementation of exception handling as well as checkpointing is in the making. It will be available together with Mads Ohm Larsen's masters thesis [3].

[1]Corresponding Author: *Mads Ohm Larsen, Datalogisk Institut, Universitetsparken 1, DK-2100, Copenhagen, Denmark*. Tel.: +45 3532 1421; Fax.: +45 3532 1401; E-mail: `omega@diku.dk`.

1. Basics

Four different kind of channel types exist: one-to-one, one-to-any, any-to-one, and any-to-any. These four types are very much alike, however only one-to-one are part of "Core CSP" as defined by Hoare [1]. The rest have to be built with the use of the interleaving operator.

In the following section i, j, n, m are all elements of $\mathbb{N}$, and $1..n$ will be used as a shorthand for the set $\{1, 2, \ldots, n\}$.

One-to-One A one-to-one channel is simply a channel with one writer and one reader. This is exactly what we have in the algebra as a communication event.

$$\begin{aligned} P &= c!x \rightarrow P' \\ Q &= c?x \rightarrow Q'(x) \\ O_2O &= P \,||\, Q \end{aligned} \tag{1}$$

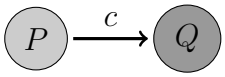


Figure 1. One-to-one channel

Any-to-One The any-to-one channel has any amount n of writers, but only one reader. This can be modelled with the algebra as many writers interleaving on a communication event. The reader and one of the writers must be ready to communicate in any order.

$$\begin{aligned} P_i &= c!x \rightarrow P_i' \\ Q &= c?x \rightarrow Q'(x) \\ A_2O &= \left(\Big|\Big|\Big|_{i \in 1..n} P_i \right) || \, Q \end{aligned} \tag{2}$$

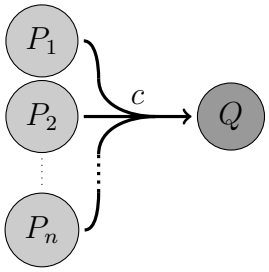


Figure 2. Any-to-one channel

One-to-Any The one-to-any channel type is equivalent to that of the any-to-one, but with the readers and writers reversed. Here we have one writer and many interleaving readers.

Any-to-Any The last channel type is the any-to-any channel. Here there are many writers and many readers, all can communicate at once.

$$\begin{aligned} P_i &= c!x \rightarrow P_i' \\ Q_j &= c?x \rightarrow Q_j'(x) \\ A_2A &= \left(\Big|\Big|\Big|_{i \in 1..n} P_i \right) || \left(\Big|\Big|\Big|_{j \in 1..m} Q_j \right) \end{aligned} \tag{3}$$

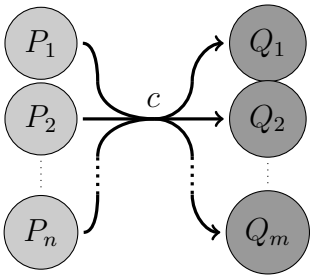


Figure 3. Any-to-any channel

At each step, one of the P_i writers gets to write to the channel and one of the Q_j readers gets to read.

Note that if $n = 1$ and $m = 1$, all we have left is:

$$\begin{aligned} P_1 &= c!x \rightarrow P_1' \\ Q_1 &= c?x \rightarrow Q_1'(x) \\ O_2O &= \left(|||_{i \in 1..n} P_i \right) || \left(|||_{j \in 1..m} Q_j \right) \\ &= P_1 \,||\, Q_1 \end{aligned} \tag{4}$$

This is identical to that of the one-to-one channel. Having either $n = 1$ or $m = 1$ gives us one-to-any and any-to-one channels respectively.

With the channels covered, we can explore the poison mechanism.

2. Poison

To poison a network is to provide a safe termination of said network [4,5]. This is done by injecting poison into the network, and having the processes propagate this poison throughout the network. In PyCSP a poisoned channel throws an exception when other processes try to communicate over it, thus poisoning the other channels.

To model a network capable of being poisoned, a supervisor process is introduced. This supervisor is listening to all the communications over a channel, be it one-to-one or any-to-any. As the communication has to be synchronised, the supervisor process can disallow communication by not engaging in the communication event.

Thus, allowing outside processes to poison the channel via a $c_{p_{id}}$ event, we can model a poisoning network like:

$$\begin{aligned} P &= (c!x \rightarrow P') \;\square\; (c_{poison} \rightarrow P_p) \\ Q &= (c?x \rightarrow Q'(x)) \;\square\; (c_{poison} \rightarrow Q_p) \\ S_{ok} &= \Big(d \;:\; \{c.m \mid m \in \alpha c\}) \rightarrow S_{ok}\Big) \;\square\; \left(\Box_{id}\, c_{p_{id}} \rightarrow S_e\right) \\ S_e &= c_{poison} \rightarrow S_e \;\square\; SKIP \end{aligned} \tag{5}$$

Note that no two other processes can have the same $c_{p_{id}}$ as that would mean that they had to agree on poisoning the c channel. P_p and Q_p are two processes that poison all of P's and Q's channels respectively. S_e is a process which will poison the processes that share c. Figure 4 shows how these processes interact.

$$P_p = \underset{c \in \alpha P}{||} c_{p_{id}} \rightarrow SKIP \tag{6}$$

To create a poisonable-network P, Q, and S_{ok} process should be run in parallel.

$$POISON = P \,||\, Q \,||\, S_{ok} \tag{7}$$

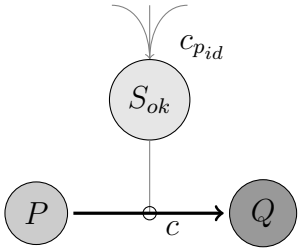


Figure 4. Poison on one-to-one channel

This one-to-one algebra of poison in equation (5) can easily be extended to any-to-any channels. The S_{ok} and S_e processes are the same, as they only concern the channel.

$$\begin{aligned} P_i &= (c!x \rightarrow P_i') \;\square\; (c_{poison} \rightarrow P_{p_i}) \\ Q_j &= (c?x \rightarrow Q_i'(x)) \;\square\; (c_{poison} \rightarrow Q_{p_j}) \end{aligned} \tag{8}$$

Again, P_{p_i} and Q_{p_j} are processes that poison all of P_i and Q_j's channels respectively like equation (6).

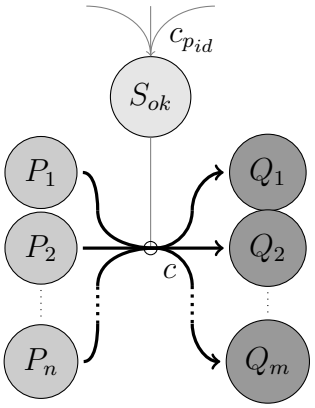


Figure 5. Poison on any-to-any channel

To create a poisonable-network we need to let all of P_i and Q_j interleave. S_{ok} should be run in parallel with these:

$$POISON_{A_2A} = \left(\underset{i \in 1..n}{|||} P_i \right) || \left(\underset{j \in 1..m}{|||} Q_j \right) || \, S_{ok} \tag{9}$$

And again, having $n = 1$ and $m = 1$ gives us

$$POISON_{O_2O} = P_1 \,||\, Q_1 \,||\, S_{ok} \tag{10}$$

With poison on any-to-any channels, we can now explore retirement, which works much like poison.

3. Retirement

Instead of poisoning a channel we can retire a process from the channel [6]. This works by letting a process decide no longer to subscribe to events on a channel c.

When modelling retirement the initial processes for P_i and Q_i, from equation (3), are the same.

$$\begin{aligned} P_i &= (c!x \rightarrow P_i') \ \Box \ (c_{poison} \rightarrow P_p) \\ Q_j &= \big(c?x \rightarrow Q_j'(x)\big) \ \Box \ (c_{poison} \rightarrow Q_p) \end{aligned} \tag{11}$$

The supervisor's S_e process is also the same, as it should tell all processes with channel c that all processes are retired.

The S_{ok} process needs to be altered to incorporate retirement. Here we give two new events, $c_{rw_{id}}$ and $c_{rr_{id}}$, to retire either a writer or a reader. As it is up to the programmer to make sure that a process P no longer writes or reads from c after it has retired, the supervisor only needs to know how many of each are subscribing to the channel in the first place.

$$\begin{aligned} S_{ok}(n,m) = \ & \text{if } (n = 0 \text{ or } m = 0) \\ & \quad S_e \\ & \text{else} \\ & \quad ((d \ : \ \{c.me \mid me \in \alpha c\}) \rightarrow S_{ok}\,(n,m)) \\ & \quad \Box \big(c_{rw_{id}} \rightarrow S_{ok}(n-1,m)\big) \\ & \quad \Box \big(c_{rr_{id}} \rightarrow S_{ok}(n,m-1)\big) \\ & \text{end} \end{aligned} \tag{12}$$

Again each of the $c_{rr_{id}}$ and $c_{rw_{id}}$ events should be unique for each processes, as multiple of these means that the processes need to agree on synchronisation. When either all of the readers or writers have left a channel, it will be poisoned. This means that a process cannot input on a channel after all the readers are retired and likewise the readers cannot get output.

All the P_i and Q_j should be interleaving as usual, but this time, the supervisor needs to know how many of them there are.

$$RETIRE_{A_2A} = \left(\underset{i \in 1..n}{|||} P_i \right) || \left(\underset{j \in 1..m}{|||} Q_j \right) || \, S_{ok}(n,m) \tag{13}$$

With the notion of the supervisor in mind, we can now move on to exception handling.

4. Exception Handling

As already written exceptions can occur in any type of software, but reliable software should be able to handle these exceptions. Hilderink describes an exception handling mechanism for a CSP library for Java, called "Communicating Thread for Java" (CTJ) [7], however this is not formalised for CSP, but rather just shown to work with the current Java implementation.

Two models are discussed: the resumption model, where the exception handler corrects the exception and returns; and the termination model, where the exception handler cleans up and terminates.

Hilderink also proposes a notation for describing the exception handling in CSP algebra, using $\vec{\Delta}$ as an exception operator [8].

$$P = Q \ \vec{\Delta} \ EH \tag{14}$$

Here the process P behaves like Q, unless there is an exception, then it behaves like EH. EH in this case will only collect the exceptions, and not act upon them.

4.1. What is an Exception?

A process that suddenly behaves as $STOP$ is often an undesirable behaviour, which we would like a way to escape from. This is where exception handling comes in action.

To understand how an exception handling mechanism works, we first need to know what an exception, or exception state, is.

A process is in an exception state if part of it has caused an error and cannot terminate. This could be a division-by-zero error, failure in hardware, or another kind of error. The process cannot continue after being in an exception state, and therefore behaves like the deadlock process $STOP$, however with an exception handling mechanism, we can interrupt the failed process, and perhaps either fix and resume; or clean up and terminate the process.

A second important thing we need to understand is when the exception handling mechanism should step in. Hilderink proposes that this is done when another process tries to communicate with the failed process. This is very similar to both poison and retire, where a process is poisoned if it tries to read from or write to a poisoned channel, and it will fit together nicely with the supervisor paradigm, used for both poison and retire. In a real-life example we want a CSP-like programming language, like PyCSP, to handle some exceptions internally, using the language's normal exception handling, but in some cases we want other processes to be aware that a process has failed.

A last important thing is that a process in an exception state, will not be able to release its channels, which means that the rest of the network cannot terminate correctly. The exception handler must therefore also be responsible for releasing the channels of the process. Different ways to shut down the network in a clean manner will be discussed.

4.2. The Exception Handling Operator

As already mentioned Hilderink proposes using $\vec{\Delta}$ as an exception operator, however CSP already offers an interrupt operator: Δ [1,9].

$$P \ \Delta \ Q \tag{15}$$

This process behaves as P, but is interrupted on the first occurrence of an event from Q. P is never resumed afterwards. It is assumed that the initial event of Q is not in the alphabet of P. Hoare describes a disaster from outside a process, as a catastrophe [1] and denotes this with a lightning bolt $\lightning \notin \alpha P$. A process that behaves as P up until a catastrophe and then behaves as Q is defined by:

$$P \ \hat{\lightning} \ Q = P \ \Delta \ (\lightning \rightarrow Q) \tag{16}$$

Roscoe continues Hoare's idea of a catastrophe, and creates a throw operator Θ for internal errors [2].

$$P \ \Theta_{x:A} \ Q(x) \tag{17}$$

Here P is interrupted by a named event x from A. Hilderink and Roscoe's two operators are very similar, in the way that they interrupt the current flow of a process, and hands the control over to another process.

With the throw operator we have a way of talking about exceptions. Exceptions is simply an event x from A which occurs when a process P enters an exception state. As mentioned

above, this could be a division-by-zero error or similar. As proposed by Hilderink, this event should occur instead of communication on a channel belonging to a process in an exception state. When it occurs this way, we can treat it as a communication event.

In a real-life example we could have multiple processes running on multiple machines. Having the exception as a communication event means that we can transfer it from one machine to another, thereby propagating the exception throughout the network letting the right process handle the exception.

4.3. Exceptions and the Supervisor

Using the same paradigm as with poison and retire (the supervisor paradigm), the exception handling mechanism can be incorporated into a network. We want the exception handler to catch all exceptions, with which it can then decide what to do. The alphabet $error$ therefore contains all errors. In this section Θ will be used as a short hand for Θ_{error}, when it is not necessary to denote the error-alphabet.

Here it is shown for a network utilising the any-to-any channel, but of course it works for the other types of channel, by setting either the amount of writers or readers, or both, to one. A writer and reader process could be expressed as P_i and Q_j

$$\begin{aligned} P_i &= (c!x \rightarrow P_i') \; \Theta \; P_{e_i} \\ Q_j &= (c?x \rightarrow Q_j'(x)) \; \Theta \; Q_{e_j} \end{aligned} \tag{18}$$

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

$$\begin{aligned} P_{e_i} &= c_{e_i} \rightarrow SKIP \\ Q_{e_j} &= c_{e_j} \rightarrow SKIP \end{aligned} \tag{19}$$

However, they could also be used to correct the problem at hand; or try and then only tell the supervisor if they failed.

Depending on which of the following exception patterns one chooses, the supervisor processes will have to be adapted to this. The S_e process could try to commend the problem, poison the rest of the network, or it might even have an exception handler of its own, which it could tell. Again, as with both poison and retire, the c_{e_i} has to be unique for that process, else multiple processes would have to agree on the error state.

With this handling of exceptions we can explore different ways of shutting down the network.

4.4. Exception Patterns

The exceptions are always "triggered" by the next process reading or writing to a channel, that the process in an exception state is subscribing to. This is the same way both poison and retirement works.

4.4.1. Fail-stop

When a process enters an exception state, it stops and all data previously sent to it will get lost. An example could be a producer, sending jobs to workers. One worker enters an exception state, and the job it was granted will get lost, without the chance of recovery.

If another process tries to communicate with the failed one, the exception should propagate though the network, until the entire network is in an exception state. This is effectively the same as the process in the exception state poisoning all of its channels.

```
from pycsp_import import *

@process
def producer(cout):
  for i in range(-2, 3):
    cout(i)

@process
def worker(cin, cout):
  while True:
    x = cin()
    cout(1.0/x)

@process
def consumer(cin):
  while True:
    print cin()

c = Channel()
d = Channel()

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

```
-0.5
-1
integer division or modulo by zero
```

Figure 6. Fail-stop in PyCSP

In Figure 6, an implementation of a small producer and worker network is shown. The workers job is to take $\frac{1}{x}$ for every x passed by the producer. Of course $\frac{1}{0}$ is undefined, so the network fails.

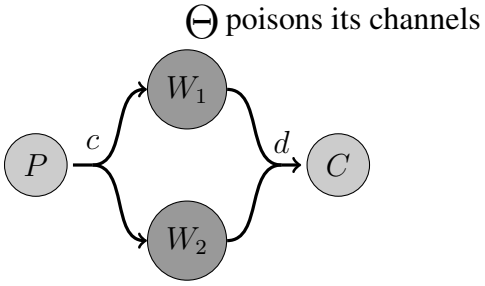


Figure 7. Fail-stop in worker process

Figure 7 shows the fail-stop network from Figure 6. The supervisor processes, which are not shown in the figure, will have to behave much like the one we saw with poisoning in equation (8), where all other processes are poisoned.

In PyCSP we have a central object where each process are created. This central object has a run-method, which is surrounded by a try-catch block. When we reach the division-by-zero, this try-catch block catches the error, runs through the process channels, and poisons each of them, thereby shutting down the network in a proper manner. Poison and retire work in the same way.

4.4.2. Retire-like Fail-stop

While fail-stop resembles poison, retire-like fail-stop mimics retire. The information sent to the processes that are in an exception state will still be lost, as with the original fail-stop. However, we have the added ability that the entire network is not shut down because of one exception. If we have a lot of distributed workers, and one fails because of e.g. a disk failure, the network will continue, but that one worker, and its job, will be lost.

4.4.3. Checkpointing

With checkpointing it is possible for a process in an exception state to roll back to the last checkpoint, which could either be defined by the programmer, or it could simply be just after the last communication with another process. That way, all information would be kept intact,

and the process at hand could try the operation that caused it to go into an exception state again. This could be a non-deterministic event, which means that it could succeed the second time around.

A counter could be attached to this form of exception pattern, which means that the process can only roll back so many times before actually failing like fail-stop, retire-like fail-stop or even broadcasting the failure. No side-effects are allowed between the last checkpoint and the point where the exception occurred, because these are operations that cannot be rolled back.

Checkpoints are quite similar to transactions, as we know them from SQL, in that we either do all the operations between two checkpoints, or none of them, because they will be rolled back.

With checkpoints the handling of the exception could be invisible to the outside world, as the roll back could happen without any other process being aware of it. This is essentially what the exceptions are meant to do, however the roll back method might not be the best way to go for it.

Remembering that PyCSP should be convenient to use, having the programmer think about checkpoints and side-effects in their code is not the way to go.

Think of the following scenario:

1. *Events up to this point*
2. Process A communicates with Process B
3. Process B receives and terminates/makes a side-effect
4. Process A goes into an exception state and wants to roll back to 1.

Process A can try to roll back the state to between the second and third item; that is, after the communication between Process A and Process B. Process B would have to roll back to its last checkpoint. If Process B has in fact terminated, Process A should enter an exception state, and possibly resolve it with fail-stop.

In the algebra, Process B wouldn't be able to terminate, before every other process was willing to do so. Therefore this is only a problem in the implementation, where we allow processes to terminate when their work is done.

4.4.4. Checkpointing algebra

Checkpointing can be modelled in the algebra with the use of a checkpoint event ⓒ [1] as well as a roll back event ⓡ. With this, we can define a new process $Ch(P)$ which behaves like P, but also incorporates checkpoints. We assume that ⓒ, ⓡ $\notin \alpha P$. To define $Ch(P)$ we need a helper $Ch2(P, Q)$ where P is the current process and Q is the most recent checkpoint. As the initial checkpoint must be the start point, we have

$$Ch(P) = Ch2(P, P) \tag{20}$$

If $P = (x : A \to P(x))$, then $Ch2(P, Q)$ is defined as

$$\begin{aligned} Ch2(P,Q) = \Big(& x : A \to Ch2(P(x), Q) \\ & \mid ⓒ \ \to Ch2(P, P) \\ & \mid ⓡ \ \to Ch(Q, Q) \Big) \ \Theta \ ⓡ \ \to Ch2(Q, Q) \end{aligned} \tag{21}$$

That is, the process P is working as usual, but upon the event ⓒ we save the current P as our checkpoint. Upon ⓡ or an error, caught by Θ, we continue on Q, which is our checkpoint.

With this checkpointing construct, it is possible to checkpoint an entire network

$$Ch(P \,||\, Q) \tag{22}$$

However, in practice, this is not what we want. We would much rather like to checkpoint each individual process

$$Ch(P) \,||\, Ch(Q) \tag{23}$$

This gives us the advantage that we can roll back each process individually. However, as already discussed, because of side-effects we cannot safely roll back over a communication. Therefore, the event ⓒ should happen after every communication. In order to do this, we need to make a change to equation (21) as the checkpoints and roll backs needs to be defined per communication, and not just one for the entire process:

$$\begin{aligned} Ch2(P,Q) = \Big(& x : A \to Ch2(P(x), Q) \\ & \Box_{c \in \alpha P} (ⓒ_c \to Ch2(P,P)) \\ & \Box_{c \in \alpha P} (ⓡ_c \to Ch2(Q,Q)) \Big) \,\Theta\, \Box_{c \in \alpha P} ⓡ_c \to Ch2(Q,Q) \end{aligned} \tag{24}$$

As the supervisor is listening to all communication, the supervisor process from equation (5) can be rewritten to:

$$\begin{aligned} S_{ok} = & \Big(d \,:\, \{c.me \mid me \in c\} \Big) \to ⓒ_c \to S_{ok} \\ & \Box \Big(ⓡ_c \to S_{ok} \Big) \end{aligned} \tag{25}$$

That is, after every communication, the supervisors tells all parties of the communication to make a synchronised checkpoint. Upon an exception, caught by Θ, they will roll themselves back as this is part of the definition in equation (24).

4.4.5. Checkpointing Examples

A small example of using the checkpointing is shown in the following network is shown in Figure 8. We want A and B to be processes which send each other a message, and forward this message to a collector C. The collector does not care about the order in which the messages are given.

A and B message each other over the same channel c, and message the collector via channel f, however, in order to do both, we need an intermediate process for both A and B called A' and B'.

$$\begin{aligned} A &= c!x \to c?y \to a!y \to A \\ A' &= a?x \to f!x \to A' \\ B &= c?x \to c!y \to b!x \to B \\ B' &= b?x \to f!x \to B' \\ C &= f?x \to C \end{aligned} \tag{26}$$

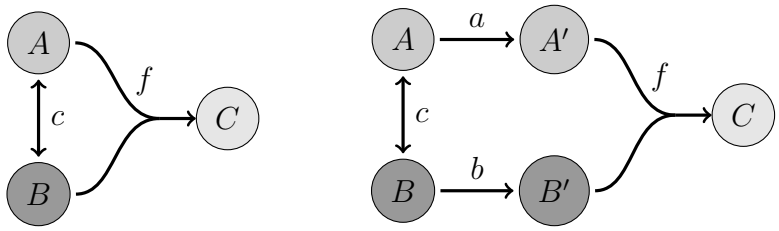
A supervisor is needed for each pair of communication events:

$$\begin{aligned} CPNet = & \Big(Ch(A) \,||\, Ch(B) \Big) \,||\, \Big(Ch(A') \,|||\, Ch(B') \Big) \,||\, Ch(C) \\ & ||\, S_{ok}(2,2) \,||\, T_{ok}(1,1) \,||\, U_{ok}(1,1) \,||\, V_{ok}(2,1) \end{aligned} \tag{27}$$

Here S, T, U and V are the supervisors, one for each channel. Therefore $c \in \alpha S$, $a \in \alpha T$, $b \in \alpha U$ and $f \in \alpha V$

We need these intermediate processes A' and B' because we want A and B to communicate, but we also want either one of A or B to communicate with C at time.

If the communication on f between B and B' fails, both are rolled back to right after the previous event. None of the other processes are affected by this.



(a) Programming model (b) CSP with intermediate processes

Figure 8. Small checkpointing example

The network in Figure 8 is implemented in PyCSP and Figure 9 shows it utilising checkpointing. This is not a working example, but rather the way we want it to work.

```
from pycsp_import import *
from random import randint

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

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

@process
def C(fin, num):
  for i in range(num):
    print i, fin()

c = Channel()
f = Channel()

Parallel(
  A(-c, +c, -f),
  B(-c, +c, -f),
  C(+f, 1000)
)
```

```
0 Ping
1 Pong
2 Ping
3 Pong
4 Ping
5 Pong
6 Ping
7 Pong
8 Ping
9 Pong
10 Ping
11 Pong
12 Ping
13 Pong
14 Ping
15 Pong
16 Ping
17 Pong
18 Ping
19 Pong
20 Ping
21 Pong
22 Ping
23 Pong
...


...
999 Pong
```

Figure 9. Checkpointing in PyCSP

5. Conclusions and Future Work

With a simple supervisor paradigm we are able to introduce exceptions in the CSP algebra, and have them work over communications. To support the supervisor paradigm, a way of visualising one-to-one, one-to-any, any-to-one, and any-to-any channels have been made. Using the supervisor together with checkpointing, we are able to roll back to previous states in pairs.

Further investigation is needed in some areas:

- A way of stopping the roll back should be devised, as explained in section 4.4.3.
  - ∗ As already discussed, this could be simply defining a explicit number of times a process is allowed to roll back, before it goes into another exception state.

- Checkpointing only works on "off" processes as described by Roscoe [10]
- A working implementation of exception handling and checkpointing using PyCSP is the topic of Mads Ohm Larsen's masters thesis [3].
- A checkpoint could be saved to disk and restored at a later time; or could be used as initial state for another identical process in another network.

Acknowledgements

Thanks go to Andrzej Filinski for his comments on this paper and contributions to the algebra.

References

[1] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
[2] A. W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.
[3] Mads Ohm Larsen. Exception Handling in Communicating Sequential Processes. To appear, aug 2012.
[4] N.C.C. Brown and P.H. Welch. An introduction to the Kent C++CSP library. In J.F. Broenink and G.H. Hilderink, editors, *Communicating Process Architectures 2003*, volume 61 of *Concurrent Systems Engineering Series*, pages 139–156, Amsterdam, The Netherlands, September 2003. IOS Press.
[5] Bernhard Sputh and Alastair R. Allen. JCSP-Poison: Safe Termination of CSP Process Networks. In *Communicating Process Architectures 2005*, pages 71–107, sep 2005.
[6] Brian Vinter, John Markus Bjø rndalen, and Rune Mø llegard Friborg. PyCSP Revisited. In Peter H. Welch, Herman Roebbers, Jan F. Broenink, Frederick R. M. Barnes, Carl G. Ritson, Adam T. Sampson, G. S. Stiles, and Brian Vinter, editors, *Communicating Process Architectures 2009*, pages 263–276, nov 2009.
[7] Gerald Henk Hilderink. Exception Handling Mechanism in Communicating Threads for Java. In Jan F. Broenink, Herman Roebbers, Johan P. E. Sunter, Peter H. Welch, and David C. Wood, editors, *Communicating Process Architectures 2005*, pages 317–334, sep 2005.
[8] Gerald Henk Hilderink. *Managing complexity of control software through concurrency*. PhD thesis, Enschede, May 2005.
[9] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
[10] A. W. Roscoe. On the expressiveness of CSP. feb 2011.