UNIVERSITY OF COPENHAGEN

Master Thesis

Mads Ohm Larsen

Exception Handling in Communicating Sequential Processes

Brian Vinter and Andrzej Filinski

17th August 2012

Abstract

Abstract

Exceptions can occur in all software, however, to be reliable, a program should be able to handle it. Exception handling has been formalised in Communicating Sequential Processes (CSP). Before doing this, the basics of channels has been investigated, and a supervisor paradigm has been created. Channels are discussed as communication events which are monitored by this 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.

Five different kinds of exception handling is discussed: fail-stop, retire-like fail-stop, broadcast, message replay, and checkpointing. Fail-stop and retire-like fail-stop works like poison and retire, when a process enters an exception state. 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. These exception patterns, as well as implicit retirement, was implemented in PyCSP.

In addition to this thesis, a paper was submitted and accepted to Communicating Process Architectures 2012, a conference on concurrent and parallel programming.

Resumé

Fejl kan opstå i al slags software, men pålidelige programmer skal kunne håndtere disse. Fejlhåndtering er blevet formaliseret i Communicating Sequential Processes (CSP). For at gøre dette, blev de grundlæggende kanaler undersøgt og et vejlederparadigme blev lavet. Kanaler bliver diskuteret som kommunikationshændelser, som bliver overvåget af denne vejlederproces. Vejlederprocesen bliver også brugt til at formalisere forgiftnings- og pensioneringshændelser. Fejlhåndtering og checkpointing bliver brugt til at komme tilbage efter en fejl. Vejlederprocesen er vital for checkpointing og tilbagerulning.

Fem forskellige slags fejlhåndtering bliver diskuteret: fail-stop, retire-like fail-stop, broadcast, message replay og checkpointing. Fail-stop og retire-like fail-stop virker som forgiftning og pensionering, når en proces går i en fejltilstand. Checkpointing virker ved at fortælle vejlederprocesen at denne skal rulle alle deltagere i en kommunikationshændelse tilbage til en tilstand lige efter deres sidste succesfulde kommunikation. Disse fejlhåndteringsmetoder, og implisit pensionering, er blevet implementeret i PyCSP.

Ud over dette speciale er der også blevet udarbejdet en artikel, som er blevet optaget på Communicating Process Architectures 2012, en konference om sideløbende- og parallelprogramering.

Contents

Glossary of Symbols

Notation	Meaning
αP	the alphabet of process P
αc	the set of messages communicable on channel c
$a \rightarrow P$	prefixing, a then P
$P;\ Q$	P (successfully) followed by Q
$P \ \vert\vert\ Q$	P parallel with Q
$P \ \vert\vert\vert\ Q$	P interleaved with Q
$a \rightarrow P \mid b \rightarrow Q$	choice, a respectively b followed by P respectively Q
$P \square Q$	deterministic choice, P or Q
$P \sqcap Q$	non-detministic choice, P or Q
$c!x$	output or send x on channel c
$c?x$	input or receive x on channel c
$(x : A \rightarrow P(x))$	choice of x from A then $P(x)$
$P \bigtriangleup Q$	P interruptible by Q
$\lightning$	catastrophe
$P \hat{\lightning} Q$	P, but on catastrophe Q
$P \ \Theta_{error}\ Q$	P, but on event from *error* Q
ⓒ	checkpoint event
ⓡ	roll back event

Chapter 1

Introduction

Exceptions can occur in any type of software, however reliable software should be able to handle these exceptions. Most programming languages offers an exception handling mechanism, in order for the programmer to specify what to do in case of an exception. These exception handling mechanisms are usually known as *throw* and *catch*. The first is a mechanism to transfer control, it is known as raise, or *throw*. The exception is said to be raised or thrown. The second mechanism, *catch*, is where control gets transferred to. An exception can be caught and the flow can continue.

Communicating Sequential Processes (CSP) is a formal language to describe concurrent systems. The first paper on CSP was published in 1978 [Hoa78] and it has evolved ever since. The *sequential* part of CSP is something of a misnomer, since CSP can handle both sequential as well as parallel processes today.

CSP is used to describe a *network* of communicating *processes*. A process is composed of two things, namely *events* and *primitive processes*. The primitive processes are fundamental behaviour, such as the deadlock process $STOP$ and the successful termination process $SKIP$. Events is the simplest construct for interaction and communication. Two parallel CSP processes capable of engaging in an event, must do so together, or in CSP terms, synchronised. A subclass of events are *communication events*. These have to be synchronised as well, but the two processes in question have got each their *end* of the communication. One process can input and the other receive the output. We say that these two processes are communicating via a *channel*, however channels are not limited to two communicating processes. Construction of channels with capability of handling communication between more than two processes are discussed in chapter 3.

Channels can be used to communicate between processes, however when a channel is not needed any more, it should be shut down. Some implementations of CSP has a *poison* construct [BW03, SA05] which can be used to safely terminate a network. Poison is discussed in chapter 4. Poisons less aggressive brother, *retirement*, is discussed in chapter 5.

Exception handling in concurrent systems are not as easy as throw and catch. Internally, in a single process in a network, this could be the case, however across multiple processes something else is needed. An exception handling mechanism for CSP are discussed in chapter 6.

Since 1978 CSP has been the basis for several programming languages. Some programming languages are directly based of of CSP, languages like *Go* and *occam*, however CSP is also the basis for many libraries for already renowned programming languages, such as JCSP for Java

[WM00], C++CSP2 for C++ [BW03, Bro07], CHP for Haskell [Bro08], and PyCSP for Python [BVA07]. This thesis will focus on the implementation in PyCSP, however a discussion of these programming languages and libraries is present in chapter 7.

In this thesis I will investigate how exception handling can be introduced in the CSP algebra as well as an implementation like PyCSP. PyCSP builds heavily upon the notion that it should be easy to create a concurrent network, get the network up and running, and equally as easy to get the data out of the network. Keeping this in mind, exception handling should be easy to use as well, with no big overhead in means of programming time. Examples of how to use these exceptions and the handling thereof will be shown in chapter 8.

In addition to this thesis, a paper (appendix A) was submitted and accepted to Communicating Process Architectures 2012, a conference on concurrent and parallel programming.

Chapter 2

Motivation

Almost every programming language has an exception handling mechanism built into the language. These are usually new scopes, where the exception will be thrown from. Exceptions can be caught in the same level scope, or be propagated up, until it is caught. If it is not caught in the programmers code, it will usually hit the interpreter or operating systems exception handling mechanism, where it will be handled. Hoares CSP [Hoa85] did not have an *internal exception operator*, however it did have the catastrophe event ↯, which should be seen as an external entity causing a catastrophe for a process. Roscoe adds upon Hoares catastrophe [Ros10], and creates a *throw* operator, which works much like our programmatic try-catch statements. That said, Roscoes throw operator is only internal, which means that each process needs to know how to handle each type of error, or else it will deadlock. PyCSP, and CSP in general, are missing a mechanism, which is able to propagate the exception between processes, maybe even letting another process handle it.

PyCSP strives to be an easy to learn and easy to use CSP-like programming library [FVB10, VBF09]. This is because the intended users of PyCSP is not computer-scientists or expert programmers, but rather all kinds of scientists. General scientists cannot be excepted to learn CSP, which is why PyCSP should be as easy to use as Python. Any new constructs should be equally as easy to use, as the rest of PyCSP. Of course the programmers should not create error prone software, but PyCSP should be able to handle errors if they occur.

Without proper exception handling a lot of work could be lost to run-time errors, especially in the field of science. A simple Monte Carlo Pi method could run for a very long time, before encountering an error. As the Monte Carlo Pi method only returns back, once it has found an approximation for π, all of the work will be lost, if it encounters an error. This is also true for exception handling with a standard `try-catch` mechanism.

The CSP exception handling mechanism should take this into account. It should be able to roll back to a last known, working, configuration. To do this, a process needs to be able to tell other processes, that it has failed, rolling them back to their last working configuration as well. Hoare describes an internal checkpointing mechanism as well as how to roll back for restartable processes. This can be used to checkpoint each process on their own.

A process in an exception state should have several options on how to proceed. Normally a programmer will state what will happen in a `catch` scope in the respective programming language. However with a CSP network we have some other options. A network could be poisoned, a process could be retired or, as mentioned above, a process could be rolled back. Internally a process could still catch the exceptions and respond in their own manner.

In order to talk about poison, retirement and exception handling in formal CSP, we need to first have an understanding of the basics of channels. This is the topic for the next chapter.

Chapter 3

Basics

In this section I will explore the basics of communication with CSP algebra [Hoa85].

Four different kind of channel types exists: 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 [Hoa85]. The rest has 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 $\{1, 2, \ldots, n\}$.

3.1 One-to-One Channels

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 as seen in equation (3.1). Figure 3.1 shows this communication visually.

$$\begin{aligned} P &= c!x \rightarrow P' \\ Q &= c?x \rightarrow Q'(x) \\ O_2O &= P \,||\, Q \end{aligned} \tag{3.1}$$

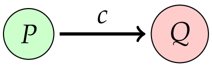


Figure 3.1: One-to-one channel

3.2 Any-to-One Channels

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. This is shown visually in figure 3.2 and the CSP algebra in equation (3.2).

$$\begin{aligned} P_i &= c!x \rightarrow P_i' \\ Q &= c?x \rightarrow Q'(x) \\ A_2O &= \left(\underset{i \in 1..n}{|||} P_i \right) || \, Q \end{aligned} \tag{3.2}$$

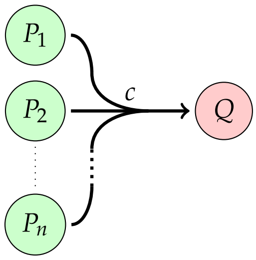


Figure 3.2: Any-to-one channel

To see that this is correct, we set $n = 2$. A_2O will then be equal to:

$$A_2O = (P_1 \,|||\, P_2) \,||\, Q \tag{3.3}$$

If we insert P_1, P_2 and Q, we can see that only one P will be able to send to Q. By using Hoares L6 law, about interleaving, we get:

$$\begin{aligned} A_2O &= \Big((c!x \rightarrow P_1') \;|||\; (c!y \rightarrow P_2') \Big) \,||\, (c?x \rightarrow Q'(x)) \\ &= \Big(c!x \rightarrow (P_1' \;|||\; (c!y \rightarrow P_2')) \;\square\; (c!y \rightarrow ((c!x \rightarrow P_1') \;|||\; P_2')) \Big) \;||\; (c?x \rightarrow Q'(x)) \end{aligned} \tag{3.4}$$

The choice is on $c!x$ and $c!y$ together with $c?x$ from Q, therefore either $c!x$ or $c!y$ will happen. Afterwards, if Q is still willing to accept communication via c, the other communication can take place, as the rest of P_1' and P_2' is interleaved with the rest of $Q'(x)$.

3.3 One-to-Any Channels

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 as shown in figure 3.3 and equation (3.5).

$$\begin{aligned} P &= c!x \rightarrow P' \\ Q_j &= c?x \rightarrow Q_j'(x) \\ O_2A &= P \,||\, \left(\underset{j \in 1..m}{|||} Q_j \right) \end{aligned} \tag{3.5}$$

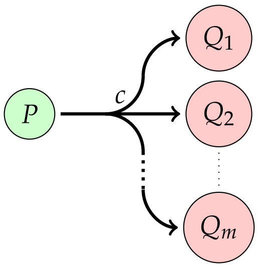


Figure 3.3: One-to-any channel

3.4 Any-to-Any Channels

The last channel type is the any-to-any channel. Many writers and many readers are able to communicate all at once. This takes the many part from both of the above and combines them as shown in figure 3.4 and equation (3.6).

$$\begin{aligned} P_i &= c!x \rightarrow P_i' \\ Q_j &= c?x \rightarrow Q_j'(x) \\ A_2A &= \left(\underset{i\in 1..n}{|||} P_i \right) \mid\mid \left(\underset{j\in 1..m}{|||} Q_j \right) \end{aligned} \tag{3.6}$$

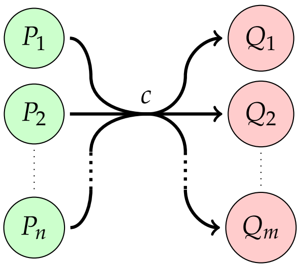


Figure 3.4: Any-to-any channel

One of the P_i writers get to write to the channel and of the Q_j readers get to read.

To show this works as it should, we set $n = 2$ and $m = 2$. This means we have P_1 interleaved with P_2 parallel with Q_1 interleaved with Q_2. Following the same example as with the any-to-one channel, we get:

$$\begin{aligned} P_1 &= c!x \rightarrow P_1' \\ P_2 &= c!y \rightarrow P_2' \\ Q_1 &= c?x \rightarrow Q_1'(x) \\ Q_2 &= c?x \rightarrow Q_2'(x) \\ A_2A &= \left(P_1 \mid\mid\mid P_2 \right) \mid\mid \left(Q_1 \mid\mid\mid Q_2 \right) \end{aligned} \tag{3.7}$$

Here $P_1 \mid\mid\mid P_2$ will work just like in the any-to-one example above, which ends with a choice of either $c!x$ or $c!y$. $Q_1 \mid\mid\mid Q_2$ however, will be different, as they both receive on channel c.

$$\begin{aligned} \left(Q_1 \mid\mid\mid Q_2 \right) &= \left(c?x \rightarrow Q_1'(x) \right) \mid\mid\mid \left(c?x \rightarrow Q_2'(x) \right) \\ &= \left(c?x \rightarrow (Q_1'(x) \mid\mid\mid Q_2) \;\square\; c?x \rightarrow (Q_1 \mid\mid\mid Q_2'(x)) \right) \\ &= c?x \rightarrow \left((Q_1'(x) \mid\mid\mid Q_2) \sqcap (Q_1 \mid\mid\mid Q_2'(x)) \right) \end{aligned} \tag{3.8}$$

That is, x is being received on channel c and then an internal choice between $Q_1'(x)$ and $Q_2'(x)$ are being made. We cannot know which process has received the message, before that process reacts, therefore the internal choice.

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

$$\begin{aligned}
P_1 &= c!x \rightarrow P_1' \\
Q_1 &= c?x \rightarrow Q_1'(x) \\
O_2O &= \left(\mathop{|||}_{i \in 1..n} P_i \right) \,||\, \left(\mathop{|||}_{j \in 1..m} Q_j \right) \\
&= P_1 \,||\, Q_1
\end{aligned} \tag{3.9}$$

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.

3.5 Buffered Channels

Before we go beyond the basics, a small discussion of the channels that have been made is in order. There are no need to extend Hoares CSP with additional channels, as it has just been shown that they can be made purely with interleaving processes.

In JCSP each of the four types are implemented individually, while PyCSP only have any-to-any channels. As I have just shown, an one-to-one channel is simply an any-to-any channel, with only one reader and one writer. PyCSP do not need these extra channels.

A writing process will always have to wait for the reading process to read, before it itself can continue. However, with buffered channels, the writing process can just pass the message along, and continue. This is true, because a buffering channel will behave as a buffering process which only job is to read on one channel and write on another.

In the following equation is a small network with a buffering process for channel c. This network is shown in figure 3.5.

$$\begin{aligned}
P &= c!x \rightarrow P' \\
Q &= c!x \rightarrow Q' \\
R &= B_{()} \\
&\quad \text{where} \\
&\qquad B_{()} = c?x \rightarrow B_{(x)} \\
&\qquad B_{(x)^\frown s} = \left(c?y \rightarrow B_{(x)^\frown s^\frown (y)} \,\square\, c'!x \rightarrow B_s \right) \\
S &= c'?x \rightarrow S'(x) \\
T &= c'?x \rightarrow T'(x) \\
BUF &= \left(P \,|||\, Q \right) \,||\, R \,||\, \left(S \,|||\, T \right)
\end{aligned} \tag{3.10}$$

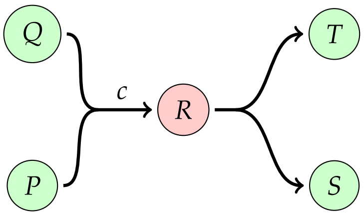


Figure 3.5: A network with a buffered channel c

Here P and Q sends their input to channel c as usual, however it is not S or T which receives at first. An intermediate process R is synchronising on this communicating, in place of S. This process either receives from its left or sends on its right, maintaining a list of messages received, but not yet sent. Interleaving the communication events will ensure the messages to be delivered in correct order, even on buffered channels.

3.5.1 Creating Buffered Any-to-any Channels Without Interleaving

Without the interleaving construct, we could still have an any-to-any-like channel. A bunch of channels would be needed, instead of just one. Each n writers and m readers would have to have a channel connected to each other, giving us a total of nm channels. This net of channels can be seen in figure 3.7.

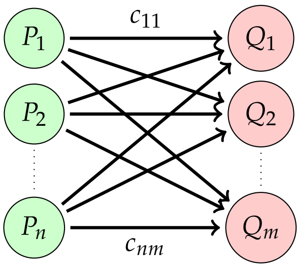


Figure 3.6: Any-to-any channels without interleaving

The algebra for such a network would be quite different from what we have seen until now.

$$\begin{aligned} P_i &= \mathop{\Box}_{j\in 1..m} (c_{ij}!x \rightarrow P_i') && \forall i \in 1..n \\ Q_j &= \mathop{\Box}_{i\in 1..n} (c_{ij}?x \rightarrow Q_j'(x)) && \forall j \in 1..m \\ AltNet &= \Big(\mathop{||}_{i\in 1..n} P_i \Big) \,||\, \Big(\mathop{||}_{j\in 1..m} Q_j \Big) \end{aligned} \tag{3.11}$$

Here every process P_i is ready to write on all of c_{i1} to c_{im} channels, which is determined by a choice. Likewise, every process Q_j is ready to read from the c_{1j} to c_{nj} channels. Note that all these processes are run in parallel.

Buffering each of these channels, would allow the messages to be reordered, thereby not going from e.g. P_1 to Q_1 in the same order they were sent. However, a giant buffering process can be inserted the same way as figure 3.5. Since all communication now runs through this one buffering process, half of the channels are moved to the other side of it, having n channels going into it, and m channels leaving it.

This network might not seem that bad, however, if we try to write a PyCSP program without using any-to-any channels, as these cannot exists without interleaving, we would end up with something along the lines of listing 3.1. Note that each of these channels are in fact any-to-any channels, but are only used as one-to-one.

Here we create the producer, a consumer and 10 workers. For this we need 20 channels. One going from the producer to each of the 10 workers, and one going from each of the 10 workers to the consumer. None of these channels are buffered, as that would complicate things even more.

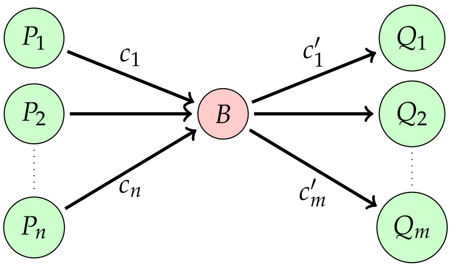


Figure 3.7: Any-to-any channels without interleaving, with a buffering process

We do an `AltSelect` on each of the channels that the producer uses, to pass the job to the first worker who is ready. The consumer also does an `AltSelect` to see which worker it needs to get a job from. This is similar to the what we saw in equation (3.11). In PyCSP we have any-to-any channels, which I have shown can be used like any-to-one and one-to-any if only one process is using one end.

In listing 3.2 the same network as before are simulated again, but this time we allow for any-to-any channels and therefore only use two channels, instead of 20.

In both listing 3.1 and 3.2 I have used a construct called `retire`. This construct will be described in the following section, but first we need to take a look at another, similar, construct, namely the `poison` construct. When we understand `poison`, we can easily modify it to `retire`.

```
from pycsp.threads import *

NUM_PROCESSES = 10

@process
def producer(cout):
  for i in range(1, 30):
    _ = AltSelect(*[OutputGuard(co, msg=i) for co in cout])

  for i in range(NUM_PROCESSES):
    retire(cout[i])

@process
def worker(cin, cout):
  while True:
    x = cin()
    cout(x*2)

@process
def consumer(cin):
  while cin:
    try:
      ch_end, x = AltSelect(*[InputGuard(ci) for ci in cin])
      print x
    except ChannelRetireException:
      if ch_end in cin:
        cin.remove(ch_end)

producerCr, producerCw, consumerCr, consumerCw = [], [], [], []

for i in range(NUM_PROCESSES):
  pc = Channel()
  producerCr.append(+pc)
  producerCw.append(-pc)

  cc = Channel()
  consumerCr.append(+cc)
  consumerCw.append(-cc)

Parallel(
  producer(producerCw),
  consumer(consumerCr),
  *[worker(producerCr[i], consumerCw[i]) for i in range(NUM_PROCESSES)]
)
```

Listing 3.1: A simple network with only one-to-one channels

```
from pycsp.threads import *

NUM_PROCESSES = 10

@process
def producer(cout):
  for i in range(1, 30):
    cout(i)
  retire(cout)

@process
def worker(cin, cout):
  while True:
    x = cin()
    cout(x*2)

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

producerC = Channel()
consumerC = Channel()

Parallel(
  producer(-producerC),
  NUM_PROCESSES * worker(+producerC, -consumerC),
  consumer(+consumerC)
)
```

Listing 3.2: A simple network with any-to-any channels

Chapter 4

Poison

To poison a network is to provide a safe termination of said network [BW03, SA05]. 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 tries to communicate with it, thus poisoning other processes.

No one has shown how channels in PyCSP work with formal CSP. In the previous chapter I showed how channels could be modelled. This should be the same for all implementations of CSP. In this section I will show how poison is handled in PyCSP using formal CSP. It is possible that other implementations have their own, and different, way of enabling poison in a network.

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 processes to poison the channel via a $c_{p_{id}}$ event, we can model a poisonable one-to-one channel like:

$$\begin{aligned} P &= \left(c!x \rightarrow P'\right) \;\square\; \left(c_{poison} \rightarrow P_p\right) \\ Q &= \left(c?x \rightarrow Q'(x)\right) \;\square\; \left(c_{poison} \rightarrow Q_p\right) \\ S_{ok} &= \left(d \;:\; \{c.m \mid m \in \alpha c\}\right) \rightarrow S_{ok}\Big) \;\square\; \left(\mathop{\square}_{id} c_{p_{id}} \rightarrow S_e\right) \\ S_e &= c_{poison} \rightarrow S_e \;\square\; SKIP \end{aligned} \tag{4.1}$$

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 poisons all of P respectively Q's channels.

$$P_p = \mathop{||}_{c \in \alpha P} c_{p_{id}} \rightarrow SKIP \tag{4.2}$$

S_e is a process which will only engage in a poison event or terminate together with the rest of the network. Figure 4.1 shows how these processes interact.

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

$$POISON = P \;||\; Q \;||\; S_{ok} \tag{4.3}$$

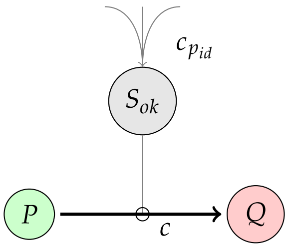


Figure 4.1: Poison on one-to-one channel

As already mentioned the network is poisoned by S_{ok} acting on an event $c_{p_{id}}$. S_{ok} will become S_e which will only interact on the event c_{poison} or $SKIP$, in the latter case, it will just terminate. The c_{poison} event will in turn let P and Q become P_p and Q_p. It will also deem the channel c unusable, as the c channel is in the alphabet of S_e.

This one-to-one algebra of poison in equation (4.1) can easily be extended to any-to-any channels, which we will see in the next section.

4.1 Combining Any-to-any Channels and Poison

Poison works on more than just one-to-one channels, in fact it works on any-to-any channels. In this section I will show how it can be extended to these channels. As described earlier, the other types can be derived from any-to-any channels by setting either $n = 1$ or $m = 1$ or both, so showing that poison works on any-to-any channels, we should be able to derive them working for both any-to-one and one-to-any channel types.

With any-to-any channels we have n writers ($P_1 \dots P_n$) and m readers ($Q_1 \dots Q_m$). These all need to be able to communicate, but the any-to-any channel should support poisoning, so a supervisor process will again overlook the channel c over which they communicate.

The S_{ok} and S_e processes are the same, as they only concern the channel.

$$\begin{aligned} P_i &= \left(c!x \rightarrow P_i'\right) \;\square\; \left(c_{poison} \rightarrow P_{p_i}\right) \\ Q_j &= \left(c?x \rightarrow Q_i'(x)\right) \;\square\; \left(c_{poison} \rightarrow Q_{p_j}\right) \end{aligned} \tag{4.4}$$

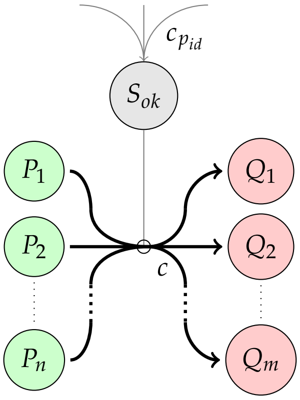


Figure 4.2: Poison on any-to-any channel

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

$$POISON_{A_2A} = \left(\underset{i \in 1..n}{|||} P_i \right) || \left(\underset{j \in 1..m}{|||} Q_j \right) || S_{ok} \tag{4.5}$$

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

$$POISON_{O_2O} = P_1 \, || \, Q_1 \, || \, S_{ok} \tag{4.6}$$

4.2 Outsider Poison

Let's look at the one-to-one poison network. If a process M, which neither reads nor writes on a channel, has a $c_{p_{id}}$ in it's alphabet, it is possible for M to poison that network, without being poisoned itself. This is not an error, but a feature of how the algebra works.

In listing 4.1 is an example of how this openness can be used in PyCSP.

```
from pycsp_import import *
import random

@process
def producer(cout):
  for i in [1,2,3,4,5]:
    cout(i)

@process
def worker(cin):
  try:
    while True:
      print cin()
  except ChannelPoisonException:
    pass

@process
def poisoner(cin):
  while True:
    if random.choice([True, False]):
      poison(+cin)
      break

c = Channel()

Parallel(
  producer(-c),
  worker(+c),
  poisoner(c)
)
```

Listing 4.1: Showing the openness of poison

The process `poisoner` never reads nor writes to the channel `c`, however, it can poison it, because it knows of `c`. The notion of outsider poison can be used to poison a network without interfering with the reader and writer counters, which are used by retirement.

The next chapter will show how retirement can used in place of poison.

Chapter 5

Retirement

5.1 Consequences of Using Poison

Using poison can have some unforeseen consequences. The main consequence is that we can poison a channel before we actually mean to. This can be seen in listing 5.1 where a producer-worker-consumer network is setup.

```
from pycsp_import import *

@process
def producer(cout):
  for i in range(1, 6):
    cout(i)
  poison(cout)

@process
def worker(cin, cout):
  while True:
    result = cin() * 2
    cout(result)

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

c = Channel()
d = Channel()

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

Listing 5.1: Poisons used with unforeseen consequences

Here the producer creates five jobs, to be taken care of by the three workers. The workers finish their job, multiplying by two, and pass along the result to a consumer. When the producer has produced all five jobs, it poisons the channel. This results in the workers being poisoned possibly before all jobs have been done, and, because a poisoned process will propagate this poison to all of its channels, the consumer might be poisoned before it has finished receiving and printing all the results.

With retirement [VBF09], this scenario would be quite different. In PyCSP we have a reader and a writer counter. When a process retires a channel, the channels read or write counter, depending on which end is being retired, is decreased. If either counter reaches zero, the channel is fully retired. This means that instead of the first poison causing the channel to be poisoned, with retirement, the last retire will cause the channel to be retired.

Listing 5.2 shows how easily poison can be swapped for retire in listing 5.1.

```
from pycsp_import import *

@process
def producer(cout):
  for i in range(1,6):
    cout(i)
  retire(cout) # poison swapped for retire

@process
def worker(cin, cout):
  while True:
    result = cin() * 2
    cout(result)

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

c = Channel()
d = Channel()

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

Listing 5.2: Retirement is the way to go

As the producer is the only writer on the `c` channel this is retired, once all jobs have been produced. When a worker tries to read from the retired channel, they will themselves retire their channels. As the `d` channel has three workers writing to it, this will not be retired before the last worker is done.

5.2 Retirement in the Algebra

When modelling retirement the initial processes for P_i and Q_i, from equation (3.6), are the same.

$$\begin{aligned} P_i &= (c!x \rightarrow P_i') \;\square\; (c_{retire} \rightarrow P_p) \\ Q_j &= \left(c?x \rightarrow Q_j'(x)\right) \;\square\; (c_{retire} \rightarrow Q_p) \end{aligned} \tag{5.1}$$

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, crw_{id} and crr_{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.

$$
\begin{aligned}
S_{ok}(n,m) = \ & \text{if } (n = 0 \text{ or } m = 0) \\
& \quad S_e \\
& \text{else} \\
& \quad ((d\ :\ \{c.me \mid me \in \alpha c\}) \rightarrow S_{ok}\,(n,m)) \\
& \quad \Box \left(c_{rw_{id}} \rightarrow S_{ok}(n-1,m)\right) \\
& \quad \Box \left(c_{rr_{id}} \rightarrow S_{ok}(n,m-1)\right) \\
& \text{end}
\end{aligned}
\tag{5.2}
$$

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

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 fully retired. 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.

$$RETIRE_{A_2A} = \left(\underset{i\in 1..n}{|||} P_i\right) \,||\, \left(\underset{j\in 1..m}{|||} Q_j\right) \,||\, S_{ok}(n,m) \tag{5.3}$$

If both $n = 1$ and $m = 1$ we have the same scenario as with poison. If either a reader or writer retires, it retires the system as one of the counters will be zero. If only one of n or m is 1, the system will be retired once every channel to that one is retired.

5.3 Openness of Retirement

As with poison, there is an openness in retirement. A process, which is neither a reader nor a writer, can retire a channel if wanted.

This was not a problem with poison, as the entire network would just shut down, but with retire this posses some different questions.

- What happens if an outsider-process retires for a reader or a writer?
  - The original process would still continue, but a channel will have been retired, and some messages would not be passed
- Should this be permitted or should the openness be closed?

The openness is generally not a problem, as the programmer decides how many processes are subscribing to a channel, when creating the initial parallelisation. If four processes are communicating on channel c, two readers and two writers, the programmer will state that S_{ok} has four spots for retirement, two for each reading and writing. Only these four processes are given the right to retire, on this channel. If more processes were given the right to retire on this channel, more retirement spots were given to begin with.

In PyCSP this is not a problem either because of the way the processes and `retire` is implemented. When a channel is prompted for a channel-end, the respectively channel counter is increased, be it a reader or a writer. This is the same in the algebra, as saying that a process has the channel in it's alphabet. That means that a process cannot know of a channel, and retire from it, unless it has already been accounted for.

We could have n and m from equation (5.3) be the number of processes with a $c_{rw_{id}}$ respectively $c_{rr_{id}}$ in their alphabet instead of just being the number of processes the programmer chooses. If we create a channel-end in PyCSP and just throw it away, we can't retire the channel, because not all readers or writers are retired.

This is similar to

$$P = SKIP \quad c_{rw_{id}} \in \alpha P$$

in the algebra.

As for the second concern, whether or not it should be closed: it should not be closed, as this is entirely up to the programmer of the network, to ensure that the processes behave in a correct manner.

5.4 Implicit Retirement

With retirement being the standard way of terminating a network, implicit retirement could be a great thing to explore. In PyCSP we have processes that work until poisoned or retired. This can be viewed in listing 5.3 and in the algebra it can be modelled as a recursive process which listens for a retire

$$W = (c?x \rightarrow W) \,\square\, (c_{retire} \rightarrow SKIP) \tag{5.4}$$

```python
@process
def worker(cin, cout):
  try:
    while True:
      x = cin() # Get the input, throw it away
  except ChannelRetireException:
    pass
```

Listing 5.3: A PyCSP worker process

If this process is neither poisoned or retired, it will work forever. However, the process producing work for this worker process (5.4) could not want to retire, but terminate instead, leaving the worker waiting forever.

$$P = c!x \rightarrow SKIP \tag{5.5}$$

Again a PyCSP example is given in listing 5.4.

```python
@process
def producer(cout, x):
  cout(x)
```

Listing 5.4: A PyCSP producer process

With these two processes run in parallel $P \,||\, W$, the network will never terminate, as the worker will never stop waiting for more data. Letting the producer retire the `cout` channel will solve this problem, however, if we use implicit retiring, and let the environment retire the channel, all channels will be retired automatically once their processes terminates. We can have implicit retirement in the algebra, by mimicking the function-decorator from PyCSP. If we have a wrapper I in the algebra, all processes that want to use implicit retirement, should be passed through that. The wrapper could be modelled simply as:

$$I(P) = P; P_r \tag{5.6}$$

where P_r is the process that retires all the channels of P. Now the parallel $I(P) \mid\mid W$ will terminate, because once finished the wrapper I will retire the channels of P.

This wrapper has been implemented in PyCSP in section 7.2.

Chapter 6

Formalising Exception Handling

Exceptions can occur in any type of software, however reliable software should be able to handle these exceptions. G. H. Hilderink describes an exception handling mechanism for a CSP library for Java, called "Communicating Thread for Java" (CTJ) [Hil05a], however this is not formalised for CSP, but rather just shown to work with the current Java implementation.

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

He also proposes a notation for describing the exception handling in CSP algebra, using $\vec{\Delta}$ as an exception operator [Hil05b].

$$P = Q \ \vec{\Delta} \ EH \tag{6.1}$$

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.

In this section I will try to formalise a exception handling mechanism, by weaving it into the already established supervisor paradigm.

6.1 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 type 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 correctly.

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, that I have 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 built-in 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 is discussed in section 6.3.

6.1.1 The Exception Handling Operator

As already mentioned Hilderink proposes using $\vec{\Delta}$ as an exception operator, however CSP already offers an interrupt operator: Δ [Hoa85, RHB97].

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

This process behaves as P but is interrupted on the first occurrence of an event of 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 [Hoa85] 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:

$$P \hat{\lightning} Q = P \, \Delta \, (\lightning \rightarrow Q) \tag{6.2}$$

Roscoe continues Hoares idea of a catastrophe, and creates a throw operator Θ for internal errors [Ros10].

$$P \, \Theta_{x:A} \, Q(x) \tag{6.3}$$

Here P is interrupted by a named event x from A. Hilderink and Roscoes 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.

6.2 Exceptions and the Supervisor

Using the same paradigm as with poison and retirement, the supervisor paradigm, the exception handling mechanism can be incorporated into a network. We want the exception handler to catch all exception, with which it can decide what to do. The alphabet *error* should therefore contain 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

$$\begin{aligned} P_i &= (c!x \rightarrow P_i') \,\Theta\, P_{e_i} \\ Q_j &= (c?x \rightarrow Q_j'(x)) \,\Theta\, Q_{e_j} \end{aligned} \tag{6.4}$$

Note, that to simplify the algebra, the poison- and retirement capabilities are not present here. The P_{e_i} and Q_{e_j} processes could be telling the supervisor that the process in hand is in an exception state.

$$\begin{aligned} P_{e_i} &= c_{e_i} \rightarrow SKIP \\ Q_{e_j} &= c_{e_j} \rightarrow SKIP \end{aligned} \tag{6.5}$$

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 exception patterns, discussed in the following sections, 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.

6.3 Exception Patterns

In sequential programs an exception is usually an escape from the current scope to another scope, where the exception can be handled. However, when working with concurrent programs, exceptions should be able to work across processes and across channels.

In this section I will look into several ways exceptions and exception handlers could exist in concurrent processes. 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 in PyCSP.

6.3.1 Fail-stop

The first way of working with exceptions, is one I will call *fail-stop*.

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

If another process tries to communicate with the failed one, or indeed on the same channel, 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.

In listing 6.1 an implementation of a small producer and worker network is shown. The workers job is to take $1/x$ for every x passed by the producer. Of course $1/0$ is undefined, so the network fails.

```
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
```

Listing 6.1: Fail-stop in PyCSP

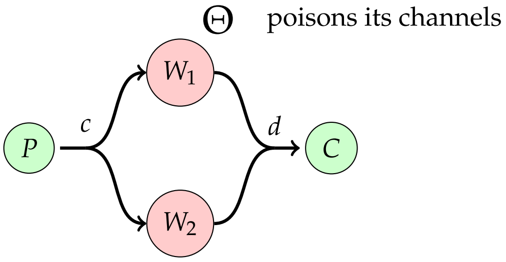


Figure 6.1: Fail-stop in worker process

Figure 6.1 shows the fail-stop network from listing 6.1. The supervisor processes, which is not shown in the figure, will have to behave much like the one we saw with poisoning in equation (4.1), 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 fail-stops each of them, thereby shutting down the network in a proper manner. This is the same way poison and retirement works.

On one hand, using this kind of exception pattern, we are able to terminate a network when one process is in an exception state. This can be useful, if it doesn't make sense to continue after a failure. On the other hand, we are not able to actually handle the exception. Handling the exception could prove important, it could be one easily handled, where it makes sense to try again, but with this pattern that is not possible.

6.3.2 Retire-like Fail-stop

The next type of exception pattern is one I will call *retire-like fail-stop*. While fail-stop resembles poison, this pattern instead mimics retire.

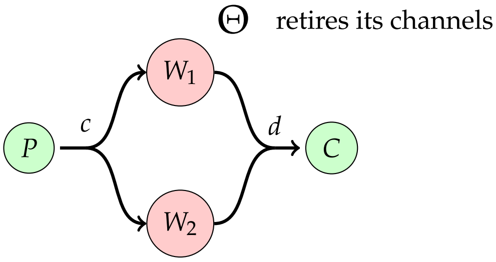


Figure 6.2: Retire-like fail-stop in one worker process

The information sent to the process 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.

This is slightly better than fail-stop, however we have no way of handling the one exception, other than ignoring it.

Both fail-stop and retire-like fail-stop have been implemented in PyCSP in section 7.3.

6.3.3 Broadcast

Looking at the retire-like fail-stop, a *broadcast* channel could be opened. When a process enters an exception state, everyone subscribing to the broadcast channel is told so. That is, if the process in exception is subscribing, the last thing it does is send a message of what went wrong over the broadcast channel. P_{e_i} from equation (6.5) could be rewritten to

$$P_{e_i} = c_{e_i} \rightarrow b_{e_i} \rightarrow SKIP$$

to incorporate this broadcast. The broadcast channel could in CSP algebra be modelled as a broadcast process, like the supervisor process already used for poison and retire.

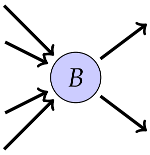


Figure 6.3: Broadcast process

Processes subscribing to the broadcast channel could decide whether or not to act upon the message, again resolving in the programmer having to make some choices. A process could be restarted this way, as the process who started the work, which went bad, could subscribe to this channel.

The job would still be lost, unless there is somewhere to figuring out which job was sent to the process in the exception state. It is not enough for the producer process to remember which jobs are sent on which channels, because the channels could be one-to-any channels. Here the producer would not know who accepted the job, and so does not know which should be resend.

Another thing worth noting here is the effects which side-effects has. Once a side-effect has occurred, one cannot just restart the job. If the worker has already sent some of the result to another process, it should not be restarted, as this would conflict with the entire result. Other side-effects include writing to file and communicating with other external entities, such as terminal, graphics card, and more.

With side-effects and the non-restartable process in mind, lets look at *message replay*.

6.3.4 Message Replay

In this exception pattern we need to be able to identify a message, as well as who the recipient of it were. Therefore each message should be wrapped in an object with an id and a recipient. The recipient of course is not known until the actual recipient has acknowledged the message.

Thus a new way of sending messages should be composed:

- Process A sends a message on an any-to-any channel
- Process B receives a message, sending back an acknowledgement to Process A
- Process A saves the message object with Process B as the recipient

This saved message object can be used later for message replay.

If Process A learns that Process B is in an exception state, the message can be replayed on the any-to-any channel, for another process to receive. This could be handled invisible to the outside world.

If the receiving process, Process B, passes on the message, it should tell Process A to forget about it. In fact, if Process B makes any kind of side-effect, Process A should no longer remember the message sent, as it is no longer guaranteed that Process B needs the same message to replay.

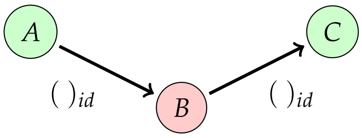


Figure 6.4: Messages with an id

One could argue that it would be up to Process B to determine whether or not it was still valid to get the message replay. In turn, it could be up to the developer whether or not it was okay.

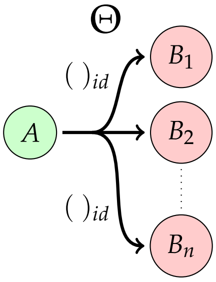


Figure 6.5: Message replay

In figure 6.5 a small network is shown. Here Process B_1 goes into an exception state, and tells the owner of the message, Process A, so. Process A, having saved the message and B_1

as the original recipient, can replay the message to the any-to-any channel once again, hoping that another process can finish the job.

Modifying the way we think of CSP communication, changing messages into actual object, might not be a good thing. Instead of having a replayable message, with ids and recipients attached, we could do checkpointing.

6.3.5 Checkpointing

With roll back checkpoints 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 thing that caused it to go into an exception state again. This could be failed hardware, or another 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 that many times, before actually failing like fail-stop, retire-like fail-stop or even broadcasting the failure. No side-effects, other than communication, are allowed between the last checkpoint and the point where the exception occurred, because these are thing that cannot be rolled back. Communications can be rolled back by propagating the roll back to the next process in the network. This way, they will “forget” that they had communicated with the process in exception, and be ready to communicate again.

Checkpoints are quite similar to transactions, as we know them from SQL, in that we do all the things between two checkpoints, or else we try and roll them back.

With roll back checkpoints the handling of the exception could be invisible for 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. Checkpointing needs to be invisible or almost invisible for the programmer.

Example

Think of the following scenario:

1. *Events up to this point*
2. Process A communicate 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. It could also try and roll back to the first item, telling Process B, if it is still alive, to roll back as well. Process B would have to roll back to just before the communication, so that the communication event can occur again. If Process B has in fact terminated, Process A should enter an exception state, and possible 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. That is, they would have to synchronise on the $SKIP$ event. Therefore this is only a problem in the implementation, where we allow processes to terminate when their work is done.

Checkpointing Algebra

Checkpointing can be modelled in the algebra with the use of a checkpoint event $\textcircled{c}$ [Hoa85] as well as a roll back event $\textcircled{r}$. With these, we can define a new process $Ch(P)$ which behaves like P, but also incorporates checkpoints. We assume that $\textcircled{c}, \textcircled{r} \notin \alpha P$. 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 say that

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

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

$$\begin{aligned} Ch2(P, Q) = \Big(& x : A \to Ch2(P(x), Q) \\ & \mid \textcircled{c} \ \to Ch2(P, P) \\ & \mid \textcircled{r} \ \to Ch(Q, Q) \Big) \ \Theta \ \textcircled{r} \ \to Ch2(Q, Q) \end{aligned} \tag{6.6}$$

That is, the process P is working as usual, but upon the event $\textcircled{c}$ we save the current P as our checkpoint. Upon $\textcircled{r}$ 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

$$Ch(P \mid\mid Q)$$

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

$$Ch(P) \mid\mid Ch(Q)$$

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 $\textcircled{c}$ should happen after every communication. In order to do this, we need to make a change to equation (6.6) as the checkpoints and roll backs needs to be defined per communication, and not just one for the entire process:

$$\begin{aligned} Ch2(P, Q) = \Big(& x : A \to Ch2(P(x), Q) \\ & \mathop{\Box}_{c \in \alpha P} (\textcircled{c}_c \to Ch2(P, P)) \\ & \mathop{\Box}_{c \in \alpha P} (\textcircled{r}_c \to Ch2(Q, Q)) \Big) \ \Theta \mathop{\Box}_{c \in \alpha P} \textcircled{r}_c \to Ch2(Q, Q) \end{aligned} \tag{6.7}$$

As the supervisor is listening to all communication, the supervisor process from equation (4.1) can be rewritten to:

$$\begin{aligned} S_{ok} = & \Big(d \ : \ \{c.me \mid me \in c\} \Big) \to \textcircled{c}_c \to S_{ok} \\ & \Box \ \Big(\textcircled{r}_c \to S_{ok} \Big) \end{aligned} \tag{6.8}$$

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 (6.7).

The implementation of checkpointing is discussed in section 7.3.3.

Chapter 7

Implementation

In this chapter I will comment on the implementation of implicit retire and the different exceptions patterns discussed in section 6.3. The entire source code is available as a branch on the original project at Google Code[1]. The files changed for this thesis can also be found appendix B in each their respective section.

7.1 CSP and CSP-like Programming Languages

As already discussed, some programming languages, like Go and occam, have their basis in CSP. Other languages, like Java, C++, Haskell, and Python have CSP libraries. Almost every programming language has an exception handling mechanism built into the language. Listing 7.1 shows how exception handling is used in Java, C++, Haskell, and Python, which are some of the programming languages with CSP libraries.

Other CSP programming languages, like Go, do not have exceptions, which are thrown and caught, build into the language. Go relies on return codes, like C code. In listing 7.2 is an example of how Go handles the same "division-by-zero" as the other languages discussed. The division of zero in line 11 creates a Go *panic*. A deferred function, `func()` is called after the return, and indeed after the panic. The error is "caught" in the `recover()` function, and checked against *nil* as this is not only called when a panic is caused, but indeed for every call to `div`.

Because occam is so much like CSP, occam does not have any form of error handling. Every program in occam is a process, and upon run-time error this process, or indeed the entire network, is shut down. The *STOP* process and run-time errors are the same for the occam compiler, which quits the program in question on *STOP*.

When implementing exception handling in PyCSP, propagating throughout a network, it should be just as easy for the programmer to see what is going on, as with normal exceptions.

7.2 Implicit Retirement

The original implementation for PyCSP does not offer implicit retirement of processes. As noted in chapter 5, this could be a great help for the audience of PyCSP.

A process that implicitly retires all of its channels, will demand less code. This will make it easier for people to grasp the content of the process, not thinking about the algebra.

---

[1] `http://code.google.com/p/pycsp/source/browse/#svn/branches/ExceptionsPyCSP`

```java
// Java
public class Example {
  public static void main(String[] args) {
    try {
      System.out.println(1/0);
    } catch (Exception e) {
      System.out.println("Cannot divide by zero");
    }
  }
}
```

```cpp
// C++
#include <iostream>
using namespace std;
int main() {
  try {
    cout << 1/0 << "\n"
  }
  catch(char * str) {
    cout << "Cannot divide by zero" << "\n";
  }
}
```

```haskell
-- Haskell
main = do
  result <- try (evaluate (1 ‘div‘ 0)) :: IO (Either SomeException Int)
  case result of
      Left ex  -> putStrLn $ "Cannot divide by zero"
      Right val -> putStrLn $ show val
```

```python
# Python
try:
  print 1/0
except:
  print "Cannot divide by zero"
```

Listing 7.1: Exception handling in Java, C++, Haskell, and Python

PyCSP already passes on a retirement. This is done in the `run()` function in the process implementation, by checking all the arguments and keyword arguments for a process for channels, retiring all of these, as seen in listing 7.3

The call to `self.fn` is what is actually running the process. Returning from this, a call to `__check_retire` can be made, which will retire all the channels, given as arguments or keyword arguments as given in listing 7.5. A modified version of `run` can be seen in listing 7.4.

Channels not given as arguments will not be retired. That is, if we create a dynamic channel inside the process, this will not get affected by implicit retirement, but then again, it would not be affected by propagation of poison or retire either.

If one of the channels are already retired, implicit retirement will still try to retire it again. This will cause the channel to throw a `ChannelRetireException` however the implementation of `__check__retire` will catch this and skip the channel, not retiring it twice.

However, if the channel is already poisoned, we should not try and retire it. This will cause all sorts of errors, so we do not even try. The `retire` functions in our channel ends have been modified to skip retiring already poisoned channels. This can be seen in listing 7.6.

```go
package main
import "fmt"

func div(x, y int) (z int) {
  defer func() {
    if err := recover(); err != nil {
      fmt.Println("Cannot divide by zero")
      z = 0
    }
  }()
  return x / y
}

func main() {
  div(1, 0)
}
```

Listing 7.2: "Exceptions" in Go

```python
# process.py
def run(self):
  try:
    # Store the returned value from the process
    self.fn(*self.args, **self.kwargs)
  except ChannelPoisonException, e:
    # look for channels and channel ends
    self.__check_poison(self.args)
    self.__check_poison(self.kwargs.values())
  except ChannelRetireException, e:
    # look for channel ends
    self.__check_retire(self.args)
    self.__check_retire(self.kwargs.values())
```

Listing 7.3: The original `run()` implementation

```python
# process.py
def run(self):
  try:
    # Store the returned value from the process
    self.fn(*self.args, **self.kwargs)
    # The process is done
    # It should auto retire all of its channels
    self.__check_retire(self.args)
    self.__check_retire(self.kwargs.values())
  except ChannelPoisonException, e:
    ...
```

Listing 7.4: An `run()` implementation offering implicit retire

7.3 Exception Patterns

In order to incorporate more than one type of exception pattern, the main `@process` decorator was modified. The `@process` decorator is now able to take optional named arguments. If this is `fail_type`, this will be used as the fail-type in the `run()` function. In the process's `__init__` function, in listing 7.7, we also check for `print_error`, `retries`, and `fail_type_after_retries`. These can be used to print the actual error, set the number of retries allowed in checkpointing (defaults to 3), and set the fail type to use after these retries. The `@process` decorator is given in listing 7.8.

The function allows for both decorators with arguments, an empty argument list, or no argument list, which means that current PyCSP programs will still run with this new version,

```
# process.py
def __check_retire(self, args):
  for arg in args:
    try:
      if types.ListType == type(arg) or types.TupleType == type(arg):
        self.__check_retire(arg)
      elif types.DictType == type(arg):
        self.__check_retire(arg.keys())
        self.__check_retire(arg.values())
      elif type(arg.retire) == types.UnboundMethodType:
        # Ignore if try to retire an already retired channel end.
        try:
          arg.retire()
        except ChannelRetireException:
          pass
    except AttributeError:
      pass
```

Listing 7.5: Implementation of `__check_retire()`

```
# channelend.py
def retire(self):
  if not self.isretired and self.channel.status != POISON:
    self.channel.leave_writer()
    self.__call__ = self._retire
    self.post_write = self._retire
    self.isretired = True
```

Listing 7.6: Skip retirement if already poisoned. This is done for both readers and writers. Only writer is shown

without changing.

7.3.1 Fail-stop

Fail-stop is implemented in much the same way as poison is in the current PyCSP implementation. To create fail-stop, I have added a new exception type, equivalent to `ChannelPoisonException`, called `ChannelFailstopException`. The `run` function has also been altered, and is shown in listing 7.9.

In this listing is also shown that we catch every exception that a function with the `@process` decorator on it will throw. As fail-stop works in the same way as poison, if a process throws an exception, it is caught by the `run` function, and handled accordingly. Unlike fail-stop the programmer is not offered a `failstop` keyword, to explicitly fail-stop a channel.

Having the fail-stop caught in the `run` function, does not mean, that you cannot catch it yourself. Like with both poison and retire, the fail-stop triggers a `ChannelFailstopException`, which can be caught in the process receiving a communication event.

7.3.2 Retire-like Fail-stop

While fail-stop is much like poison, retire-like fail-stop is like retire. With retire-like fail-stop, the network is not dead upon failure. If a process makes an error, we simply retire, instead of poison, all of its channels.

In listing 7.10 is the `run` function, but this time, it has both fail-stop and retire-like fail-stop to worry about. This is done in the last `except` clause, where we check how we should act upon failure.

```
# process.py
def __init__(self, fn, options, *args, **kwargs):
  threading.Thread.__init__(self)
  self.fn = fn

  self.fail_type = None
  if options is not None and 'fail_type' in options:
    self.fail_type = options['fail_type']

  self.args = args
  self.kwargs = kwargs

  # Create unique id
  self.id = str(random.random())+str(time.time())

  self.options = options
  self.vars = {}

  self.print_error = False
  if options is not None and 'print_error' in options:
    self.print_error = options['print_error']

  self.max_retries = CHECKPOINT_RETRIES
  if options is not None and 'retries' in options:
    self.max_retries = options['retries']

  self.retries = 0

  self.fail_type_after_retries = self.__check_retirelike
  if options is not None and 'fail_type_after_retries' in options:
    if options['fail_type_after_retries'] == FAILSTOP:
      self.fail_type_after_retries = self.__check_failstop
```

Listing 7.7: Process's `__init__` function is modified to take optional arguments

Again, as with fail-stop, we catch the exception and run the `retirelike` function on all channels given as arguments to the process. These will get retired, and will count as retired in the sense that we can retire other channels in the regular way, and still reach a reader or writer counter of zero, thereby fully retiring the channel.

7.3.3 Checkpointing

Compared to fail-stop and retire-like fail-stop, checkpointing is a chapter on its own. In order to checkpoint in PyCSP we need the following:

- A way of loading variables in a process.
- A way of saving variables in a process.
- A way of telling other processes to roll back, once the current one has encountered an error.

Each of these will have their own subsection and I will then built one atop another and unite them in the end.

Loading Variables

Loading variables should happen from a previous checkpoint. If we are at the very start of the process, a checkpoint does not exist. Therefore a method for setting default values to the variables should be incorporated into the `load_variables()` function.

```
# process.py
def process(func=None, **options):
  """
  @process decorator for creating process functions

  >>> @process
  ... def P():
  ...     pass

  >>> isinstance(P(), Process)
  True

  Processes can have a 'fail_type'.
  This is checked when failing.

  >>> @process(fail_type=FAILSTOP)
  ... def P():
  ...     1/0
  """
  if func != None:
    def _call(*args, **kwargs):
      return Process(func, options, *args, **kwargs)
    return _call
  else:
    def _func(func):
      return process(func, **options)
    return _func
```

Listing 7.8: The new `@process` decorator

```
# process.py
def run(self):
  try:
    ...
  except ChannelRetireException, e:
    # look for channel ends
    self.__check_retire(self.args)
    self.__check_retire(self.kwargs.values())
  except ChannelFailstopException:
    self.__check_failstop(self.args)
    self.__check_failstop(self.kwargs.values())
  except Exception as e:
    print e
    self.__check_failstop(self.args)
    self.__check_failstop(self.kwargs.values())
```

Listing 7.9: The `run` function with added fail-stop

In listing 7.11 is a PyCSP process, which loads variables and print out the sum.

An intermediate version of `load_variables()` is given in listing 7.12.

Here we look at all the arguments given to `load_variables()` and put their value into an array, in the same order. Each argument must be a tuple or a list of at least two elements. Ordering is important, which is why we cannot have the beauty in listing 7.13, because Python doesn't guarantee the order in hashes.

We can however cheat a bit, if we know we only wish to load a single variable.

With the `load` function from listing 7.14 we can now load each variable individually with e.g. `load(x = 1)` instead of `load_variables(('x', 1))`.

The `load_variables()` from listing 7.12 returns the variables in the same order it was given. This is not very useful on its own, however, when we actually load the variables from a checkpoint, it will be.

```
# process.py
def run(self):
  try:
    ...
  except ChannelFailstopException:
    self.__check_failstop(self.args)
    self.__check_failstop(self.kwargs.values())
  except ChannelRetireLikeFailstopException:
    self.__check_retirelike(self.args)
    self.__check_retirelike(self.kwargs.values())
  except Exception as e:
    print e
    fail_type_fn = None
    if self.fail_type == FAILSTOP:
      fail_type_fn = self.__check_failstop
    elif self.fail_type == RETIRELIKE:
      fail_type_fn = self.__check_retirelike

    if fail_type_fn is not None:
      fail_type_fn(self.args)
      fail_type_fn(self.kwargs.values())
```

Listing 7.10: `run` function, with fail-stop and retire-like fail-stop

```
@process
def P():
  x, y = load_variables(('x', 1), ('y', 2))
  print x + y # => 3
```

Listing 7.11: Loading variables and printing the sum

```
# process.py
def load_variables(*pargs):
  var = []
  for __x in pargs:
    var.append(__x[1])

  if len(var) == 1:
    return var[0]
  else:
    return var
```

Listing 7.12: First version of `load_variables()`

To load the variables from a checkpoint, some traceback-manipulation is used. `load_variables()` is a global function, however, only the `Process` object knows about `P()`'s saved variables. As it is always a `Process` object, which calls the `load_variables()` function, we can look up the call stack, and retrieve the `Process` object. The `Process` object has a variable, called `vars` which we will get back to. This traceback-manipulation in the `load_variables()` function can be seen in listing 7.15.

`loaded_vars` is a hash which contains all the variables loaded from the `Process`-object. We shall see how this is saved in a bit. In listing 7.15 we check if each variable we want to load is in this hash, or if we should take the default value, given as an argument. Unpacking the array returned in the process `P()`, we can see why it works in listing 7.11.

Notice however, that this comes a price of readability and ease of use. We have to declare our variables “twice” and in a rather peculiar way. Loading variables, we can’t reuse variables, e.g. for loop counter, as seen in listing 7.16.

Another thing worth noting is a modified way of using the for-loops, which is also shown in

```
@process
def P():
  x, y = load_variables(x = 1, y = 2) # Sadly not possible
  print x + y # => 3
```

Listing 7.13: `load_variables()` with keyword arguments is not possible, due to the ordering of arguments

```
def load(**kwargs):
  if len(kwargs) > 1:
    raise AttributeError

  for __x, __v in kwargs.iteritems():
    return load_variables((__x, __v))
```

Listing 7.14: `load` implementation

```
# process.py
def load_variables(*pargs):
  stack = inspect.stack()

  try:
    process_ = stack[3][0].f_locals
  finally:
    del stack

  loaded_vars = process_['self'].vars

  var = []
  for __x in pargs:
    if __x[0] in loaded_vars:
      var.append(loaded_vars[__x[0]])
    else:
      var.append(__x[1])

  if len(var) == 1:
    return var[0]
  else:
    return var
```

Listing 7.15: Traceback-manipulation in `load_variables()`

listing 7.16. As we might want to save the `i` variable, we cannot have `for i in range(-10, 10)`, as this would mean, that `i` would be set to `-10` at the beginning of the loop. Instead, we set `i` before, and update it, so the for-loop now reads `for i in range(i, 10)`. This means that `i` is set to itself at the beginning of the loop, and are then counted to `10`.

Saving Variables

Before we can load the variables, we need to actually save them, or else, `load_variables()` would just return its arguments. As we saw in the algebra for checkpointing, section 6.3.5, saving the variables is the job of the channel, or at least, it is the supervisors job to make sure every process checkpoints after each communication. There is no supervisor process in PyCSP, however the channels are more object-like than in the algebra, so these can easily handle the checkpointing themselves.

A channel does not know which process it belongs to, which poses a slight problem for saving variables. Like with `load_variables()` we can take into account that the channel

```
from pycsp_import import *

@process
def W():
  i = load(i = -10))
  for i in range(i, 10):
    print i

  print "pause"

  for i in range(i, 10):
    print i

Sequence(W())
```

```
-10
-9
-8
-7
-6
-5
-4
-3
-2
-1
0
1
2
3
4
5
6
7
8
9
pause
9
```

Listing 7.16: Cannot reuse variables

is only ever called inside a process. Again the call stack is brought forward, and we pick out both the process object and the actual process function, that the programmer has defined. The process function will have the variables that needs to be saved. The process object has a variable, `vars`, which hold a dictionary of variables, the very same that we loaded from in the previous section.

Listing 7.17 shows the implementation of the `save_variables()` function. The `vars` variable is set to the `locals` dictionary we pick out of the process-function.

```
# channel.py
def save_variables(self):
  stack = inspect.stack()

  try:
    locals_ = stack[2][0].f_locals
    process_ = stack[3][0].f_locals
  finally:
    del stack

  process_['self'].vars = locals_
```

Listing 7.17: Saving variables in the channel object

For each communication, the processes involved needs to save their variables. In PyCSP channels are one-way channels. That is, the same end cannot be used for both reading and writing, because we use channel-ends to determine the count for retirement. Luckily both types of channel-ends extends a uniform channel. This channel class has the functions for both reading and writing. After a successful read or write, we need to save all variables, using the `save_variables()` function.

Listing 7.18 and 7.19 shows how `save_variables()` is called after a successful read or write.

```
# channel.py
def _read(self):
  self.check_termination()
  req = ChannelReq(ReqStatus(), name = self.name)
  self.post_read(req)
  req.wait()
  self.remove_read(req)

  if req.result == SUCCESS:
    self.save_variables()
    return req.msg

  self.check_termination()

  print 'We should not get here in read!!!', req.status.state
  return None
```

Listing 7.18: A read from the channel

```
# channel.py
def _write(self, msg):
  self.check_termination()
  req = ChannelReq(ReqStatus(), msg)
  self.post_write(req)
  req.wait()
  self.remove_write(req)

  if req.result == SUCCESS:
    self.save_variables()
    return

  self.check_termination()

  print 'We should not get here in write!!!', req.status
  return
```

Listing 7.19: A write to the channel

Roll back

Now we have the ability to load and save variables. We save the variables after each communication, between two processes.

When a process in the network fails, the next process sharing a channel with that process should roll back, instead of their next communication on that channel.

```
# channel.py
def check_termination(self):
  if self.status == POISON:
    raise ChannelPoisonException()
  elif self.status == RETIRE:
    raise ChannelRetireException()
  elif self.status == FAILSTOP:
    raise ChannelFailstopException()
  elif self.status == RETIRELIKE:
    raise ChannelRetireLikeFailstopException()
  elif self.status == CHECKPOINT:
    self.status = NONE
    raise ChannelRollBackException()
```

Listing 7.20: The `check_termination` implementation

At the beginning and end of every communication, we call `check_termination`, to see whether the channel has been poisoned, retired, fail-stopped, or retire-like fail-stopped. In listing 7.20 we also check whether the channel is in a checkpoint mode. This status should be used, when we want a reader or writer to roll back, instead of communication. Like fail-stop and retire-like fail-stop, we throw an exception, if the channel is in a checkpoint mode. This exception is caught in the `run` function in Process, as seen in listing 7.21.

```
# process.py
def run(self):
  try:
    ...
  except ChannelRollBackException:
    # Another process sharing a channel with this one
    # has rolled back, so we must as well.
    self.run()
  ...
```

Listing 7.21: `run` offering to catch `ChannelRollBackExceptoin`

Unlike poison, retire, fail-stop, retire-like fail-stop, we do not want the roll back to propagate throughout the network. Therefore, we have no `__check_rollback()` function, like we have with these others. Instead we just rerun the `run` function. `load_variables` will load the variables from the last checkpoint, when we rerun the process.

The process that fails should set a roll back flag on the channels it uses. This is done when an arbitrary exception is caught from within the `run` function. The bottom except can be seen in listing 7.22.

```
# process.py
def run(self):
  try:
    ...
  except Exception as e:
    if self.print_error:
      print e

    fail_type_fn = None
    rerun = False

    if self.fail_type == FAILSTOP:
      fail_type_fn = self.__check_failstop
    elif self.fail_type == RETIRELIKE:
      fail_type_fn = self.__check_retirelike
    elif self.fail_type == CHECKPOINT:
      if self.max_retires != -1 and self.retries >= self.max_retires:
        fail_type_fn = self.fail_type_after_retires
      else:
        rerun = True
        fail_type_fn = self.__check_checkpointing

    if fail_type_fn is not None:
      fail_type_fn(self.args)
      fail_type_fn(self.kwargs.values())

    if rerun:
      self.retries += 1
      self.run()
```

Listing 7.22: `except` in `run` function

Here we call `__check_checkpointing` which sets the status of all the channels given in arguments to `CHECKPOINT`.

With this we are able to create processes which can be checkpointed and rolled back.

Chapter 8

Examples

8.1 Implicit Retirement

With implicit retirement, we do not need to retire a process any more, as this is done automatically. This means we are able to write shorter programs, with more precise processes, without having to think about closing down the network. It is still possible to retire a channel explicitly, and thereby shutting down a network.

In listing 8.1 two example usage of the implicit retirement is shown. Here we pass along a string to a waiting process. The waiting process would normally wait for the next string to be passed, but the `do_nothing` process automatically retires, because it is done.

```python
from pycsp_import import *

@process
def do_nothing(cout):
  cout("Doing")
  cout("nothing")

@process
def waiting(cin):
  while True:
    x = cin()
    print x

c = Channel()

Parallel(
  do_nothing(-c),
  waiting(+c)
)
```

```python
from pycsp_import import *

@process
def producer(cout):
  while True:
    cout("Foo")

@process
def worker(cin):
  x = cin()
  print x

c = Channel()

Parallel(
  producer(-c),
  worker(+c)
)
```

Listing 8.1: Implicit retirement

Of course one can still use the `try-except` pattern, to catch the retirement, and e.g. print the end result.

8.1.1 Monte Carlo Pi

To show the use of the `try-except` pattern, a Monte Carlo method for calculating π has been devised.

The Monte Carlo method is a method of probability. With enough data, we can say something about the thing we are trying to calculate with probability.

When calculating π it is important to remember what π is. For a circle's area we have the simple, and well known formula $A = \pi r^2$. But this means that $\pi = \frac{A}{r^2}$, so all we need to know is the area of a circle and its radius to calculate π.

If we take a dartboard and a very poor darts player (plays randomly), the number of darts that hit within the dartboard is proportional to it's area. In other words

$$\frac{\text{\#darts hitting dartboard}}{\text{\#darts hitting circumscribing square}} = \frac{\text{area of dartboard}}{\text{area of circumscribing square}}$$

Knowing this, we can calculate π like:

$$\begin{aligned}\frac{\text{\#darts hitting dartboard}}{\text{\#darts hitting circumscribing square}} &= \frac{\pi r^2}{4r^2} = \frac{\pi}{4} \\ 4\frac{\text{\#darts hitting dartboard}}{\text{\#darts hitting circumscribing square}} &= \pi\end{aligned}$$

If we want to, we can look at only the first quadrant of our coordinate system. This would mean that the darts player only hit in the first quadrant.

I will sketch out how this works in several forms:

- Non-concurrent algorithm
- CSP algebra
- PyCSP code example

Non-concurrent algorithm Our darts player hits randomly, but consistently, in the first quadrant of the dartboard. With 1 dart, we can estimate π as either 0 or 4 depending whether he hit the dartboard or not. The more darts he throw, the better estimate of π.

The algorithm is described in algorithm 1.

Algorithm 1 Monte Carlo method algorithm for calculating π

```
1. $hits = 0$
2. For 1 to $desired\ number\ of\ iterations$
  3. $x = \texttt{random}, y = \texttt{random}$
  4. Calculate $dist = x^2 + y^2$
  5. $hits = hits + 1$ if $dist < 1$
6. $\pi \approx 4\frac{hits}{desired\ number\ of\ iterations}$
```

Here `random` is a random number between 0 and 1.

CSP algebra Looking at the Monte Carlo method with CSP goggles, we need to make it more parallel. This could be done by having a number of worker processes, doing the sequential work from before, and then averaging their results in a consumer process.

The producer process could look like:

$$\begin{aligned}P(0, \_) &= SKIP \\ P(n, m) &= (c!(m) \rightarrow P(n-1, m))\end{aligned}$$

The workers will need to grab the input from P on the c channel. This input should be how many time we need to randomise and calculate whether it was a hit. We then need to collect all of these, pass them to the consumer, and go back to being a worker.

$$W_i = \left(c?m \rightarrow W_i'(m,m,0)\right)$$

$$\begin{aligned} W_i'(0,n,h) &= d!\left(\frac{4h}{n}\right) \rightarrow W_i \\ W_i'(m,n,h) &= mHits(m)?x \rightarrow W_i'(m-1,n,h+x) \end{aligned}$$

The last process that we need to define is the consumer process. This should collect all the results from all the workers, and, when retired, print this result.

$$C(h,l) = (d?x \rightarrow C(h+x,l+1)) \;\square\; (d_{retire} \rightarrow print!h \rightarrow SKIP)$$

Running these three processes in parallel, with suitable defaults, will yield an approximation to π.

$$Pi = \Big(\mathop{|||}_{i\in 1..30} I_{W_i}(W_i)\Big) \,||\, \Big(I_P(P(10000,1000)) \,||\, C(0,0)\Big) \tag{8.1}$$

Here we start 30 workers and let the producer start 10000 jobs. Each job is an integer, 1000, for which the worker calculates that many hits. The consumer collects it all and prints π.

Notice that with the use of I, none of the processes has to retire their channels.

To be fair, we also need to run two supervisor processes, in order to handle the c channel and the d channel. Equation (8.1) should be changed to

$$Pi = \Big(\mathop{|||}_{i\in 1..30} I_{W_i}(W_i)\Big) \,||\, \Big(I_P(P(10000,1000)) \,||\, C(0,0)\Big) \,||\, S_{ok}\,(1,30) \,||\, T_{ok}\,(30,1)$$

This network is shown in figure 8.1.

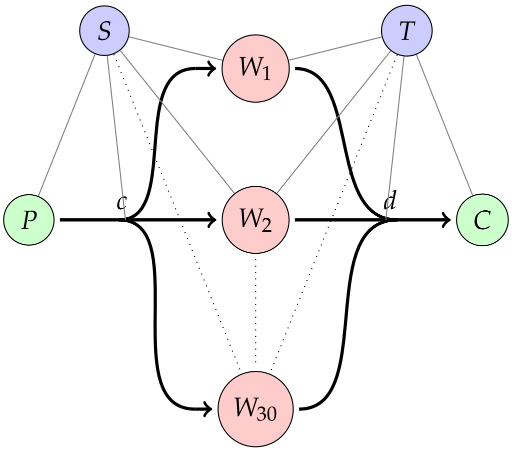


Figure 8.1: Monte Carlo Pi network

PyCSP code example Implicit retirement in PyCSP could be used to achieve briefer code, with no worries about retirement. In listing 8.2 a Monte Carlo method implementation which utilises implicit retirement is shown. The `producer` does not retire explicit, as this is now handled by PyCSP. The `consumer` still catches the retirement in order to print the result.

```python
from pycsp_import import *
from random import random

@process
def producer(cout):
  for i in range(10000):
    cout(1000)

@process
def worker(cin, cout):
  while True:
    cnt = cin()
    sum = reduce(lambda x, y: x + (random()**2 + random()**2 < 1.0),
                 range(cnt)) # Calc dist
    cout(4.0 * sum / cnt)

@process
def consumer(cin):
  cnt, sum = 0, 0

  try:
    while True:
      sum = sum + cin()
      cnt += 1
  except ChannelRetireException:
    print 'Result:', sum / cnt
    # Upon retirement, we are done and print result

jobs = Channel()
results = Channel()

Parallel(
  producer(-jobs),
  30 * worker(+jobs, -results),
  consumer(+results)
)
```

Listing 8.2: Monte Carlo Pi simulation

8.2 Exception Handling

8.2.1 Fail-stop

The implementation of fail-stop is given in section 7.3.1.

In this example, we shall look at an exception. A network is created to calculate $\frac{1}{x}$ for x going from -10 to 10. In this series is of course the division $\frac{1}{0}$ which is undefined. Python will throw an `ZeroDivisionError` exception, and would usually quit, or if caught, follow along the flow.

With the implementation of fail-stop we will instead transmit the exception via the channel. The receiving process will throw a `ChannelFailstopException` once it reads from the dead channel.

Figure 8.2 shows this network visualised with three worker processes. Listing 8.3 shows the implementation and output of this network. This contains the producer, three workers, and the consumer process. The producer communicates the numbers from -10 to 10 one at a time over the `c` channel. The workers sends $\frac{1}{x}$ on the `d` channel. Lastly the consumer prints the result. If the consumer throws the exception, it print a message, saying that it caught an exception.

We see in the output in listing 8.3 that the consumer gets every job from -10 up to 1 before it quits with the error message. This is because we are not guaranteed the same order of jobs, when working this way. The `float division by zero` comes from the implementation of fail-stop in PyCSP, where we print the exception, when it occurs. This actually comes from the worker process that dies, because we catch it in the consumer. Had we not caught it, we would not get the message twice, because we catch the `ChannelFailstopException` in listing 7.9 on page 35.

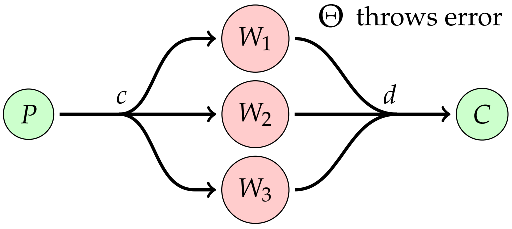


Figure 8.2: Fail-stop network

```
from pycsp_import import *

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

@process(fail_type = FAILSTOP,
         print_error = True)
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
float division by zero
Caught the exception
```

Listing 8.3: A failstop captured by the consumer and the output

8.2.2 Retire-like Fail-stop

The implementation of retire-like fail-stop is shown in section 7.3.2.

Retire-like fail-stop can be used in networks, were a one or more nodes can be retired, because of an error. If we look at the Monte Carlo Pi example from section 8.1.1, a single process's result will not make the total result much different Of course, with the Monte Carlo Pi algorithm, it might be better to just restart that one failing process. If, however, that is not possible, or the problem is persistent, retire-like fail-stop can be used.

A persistent problem could be failed hardware. If we imagine that each process is located on its own machine, or indeed on the same machine, but using different hardware, or perhaps USB devices, we can think of a network where retire-like fail-stop will come in handy.

Such a network, could be the one in figure 8.3. Here we have a producer, P, a worker, W and a consumer C, as usual. We also have a fail-process, F. This process fails after its first pass. The producer will hand jobs, here integers, to the fail-process. The fail-process, as well as the workers, job is to multiply it by two, and pass it on. The worker is latent. It isn't started with the rest of the network, but is waiting for a start signal from the producer. In a real-world scenario, the fail-process would do the task at hand on e.g. the GPU, and the normal worker on the CPU. As the GPU might be better or faster, we want all the jobs run here. If the GPU for some reason is broken, we let the CPU process take over. This network is sketched out in listing 8.4 and its output is in listing 8.5.

Using formal algebra, this network would look like:

$$\begin{aligned}
P_0 &= P'_0 = SKIP \\
P_x &= c!x \rightarrow P_{x-1} \ \Theta \ P'_x \\
P'_x &= d!x \rightarrow P'_{x-1} \\
F &= c?x \rightarrow f!(x \cdot 2) \rightarrow F \\
W &= d?x \rightarrow f!(x \cdot 2) \rightarrow W \\
C &= f?x \rightarrow print!x \rightarrow C
\end{aligned} \tag{8.2}$$

$$Rnet = \Big(I(P_{10}) \,||\, \big(I(F) \,|||\, I(W)\big) \,||\, I(C) \Big) \,||\, S_{ok}(1,1) \,||\, T_{ok}(1,1) \,||\, U_{ok}(2,1) \tag{8.3}$$

where S, T and U are the supervisor processes for the channels c, d and f respectively, and I is the implicit retire wrapper from section 5.4.

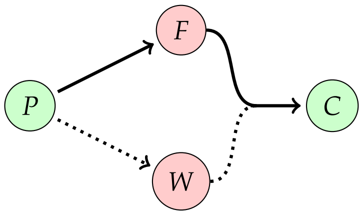


Figure 8.3: Retire-like fail-stop network with a failing hardware process

8.2.3 Checkpointing

A small example of using the checkpointing is shown in figure 8.4. We want A and B to be two processes which sends each other a message, and forwards 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'.

```
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 ChannelRetireLikeFailstopException:
      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:
    try:
      x = finish()
      print x
    except ChannelRetireLikeFailstopException:
      pass

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

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

Listing 8.4: Retire-like fail-stop network with a failing hardware process

$$
\begin{aligned}
A &= c!(''Ping'') \rightarrow c?y \rightarrow a!y \rightarrow A \\
A' &= a?x \rightarrow f!x \rightarrow A' \\
B &= c?x \rightarrow c!(''Pong'') \rightarrow b!x \rightarrow B \\
B' &= b?x \rightarrow f!x \rightarrow B' \\
C &= f?x \rightarrow print!x \rightarrow C
\end{aligned}
\tag{8.4}
$$

A supervisor is needed for each pair of communication events:

$$
\begin{aligned}
CPNet = \Big(Ch(A) \,||\, Ch(B)\Big) \,||\, \Big(Ch(A') \,|||\, Ch(B')\Big) \,||\, Ch(C) \\
||\, S_{ok}(2,2) \,||\, T_{ok}(1,1) \,||\, U_{ok}(1,1) \,||\, V_{ok}(2,1)
\end{aligned}
\tag{8.5}
$$

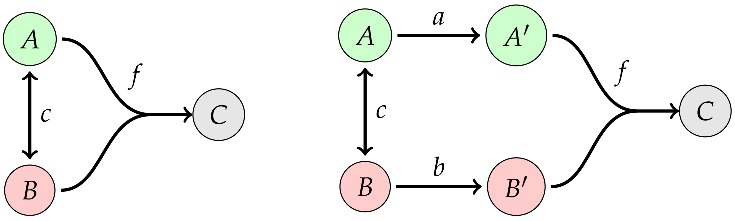
Here S, T, U and V are the supervisors, one for each channel. Therefore $c \in \alpha S$, $a \in \alpha T$, $b \in \alpha U$ and $f \in \alpha V$

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.

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

Listing 8.5: Output for listing 8.4

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 8.4: Small checkpointing example

The network in figure 8.4 is implemented in PyCSP and listing 8.6 shows it utilising checkpointing.

Another Example of Checkpointing Another way that checkpointing can be used in PyCSP is showed in listing 8.7. Here, we send twice on channel `c`, and receive twice, before printing the result. Between the two inputs, we can fail. In the listing, this is showed again as a `ZeroDivisionError`, however this could be anything. If we fail between the two sends, the first one is run again, as we load the checkpoint and restart the process.

$$\begin{aligned}
P_0 &= SKIP \\
P_x &= c!(''x:'' + x) \rightarrow c!(''y:''x) \rightarrow P_{x-1} \\
C &= c?x \rightarrow c?y \rightarrow print(x,y) \rightarrow C \\
DoubleCheck &= Ch(P) \,||\, Ch(C) \,||\, S_{ok}(1,1)
\end{aligned} \tag{8.6}$$

```
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,
         retires = -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_variables(('i', 1))
  for i in range(i, num):
    print i, fin()
  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
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
...








...
99 Pong
```

Listing 8.6: Checkpointing in PyCSP

```
from pycsp_import import *
from random import randint

@process(fail_type=CHECKPOINT,
         retries=-1)
def producer(job_out, start, end):
  i = load(i = start)
  for i in range(i, end):
    job_out("x: " + str(i))
    1 / randint(0, 1)
    job_out("y: " + str(i))

@process(fail_type=CHECKPOINT,
         retries=-1)
def consumer(job_in):
  while True:
    x = job_in()
    y = job_in()
    print x, y

c = Channel()

Parallel(
  producer(-c, -5, 6),
  consumer(+c)
)
```

```
: -5 y: -5
x: -4 y: -4
x: -3 y: -3
x: -2 y: -2
x: -1 y: -1
x: 0 y: 0
x: 1 y: 1
x: 2 y: 2
x: 3 y: 3
x: 4 y: 4
x: 5 y: 5
```

Listing 8.7: Checkpointing with multiple input on channel `c`

Chapter 9

Future Work

Exception handling in PyCSP is in a working state, however some things needs further investigation. This chapter will describe what can be done in the future.

9.1 Nonlocal

The implementation for checkpointing suggested in this thesis rely on the use of Pythons `inspect` module. The `inspect` module lets the programmer inspect the call stack, retrieving the frame containing the process when calling e.g. `save_variables`. This is not a reliable solution, as the `inspect` module do not work in the same way in every implementation of Python. While programming for this thesis Python 2.7.1 (CPython) for Mac OS has been used. Python 3 comes with a new keyword `nonlocal`, which might be used instead of getting the current frame for the process.

The following quote comes from the Python documentation of `nonlocal`:

> "The `nonlocal` statement causes the listed identifiers to refer to previously bound variables in the nearest enclosing scope. This is important because the default behavior for binding is to search the local namespace first. The statement allows encapsulated code to rebind variables outside of the local scope besides the global (module) scope."

9.2 "On" Processes

When I defined the checkpointing function in section 6.3.5, an assumption was made about the processes. The processes have to be on the form

$$P = (x : A \rightarrow P(x)) \tag{9.1}$$

If the processes is not on this form, the function $Ch2(P, Q)$ cannot be made. This is because we are not allowed to copy and "on" process, as described by Roscoe [Ros11]. Lets say we have two processes P and Q

$$\begin{aligned} P &= c \rightarrow \left(a \rightarrow STOP \sqcap b \rightarrow STOP\right) \\ Q &= c \rightarrow a \rightarrow STOP \sqcap c \rightarrow b \rightarrow STOP \end{aligned} \tag{9.2}$$

Because non-determinism is distributive, by Hoares L4 law on non-determinism [Hoa85], P and Q are equivalent. However, if P is checkpointed after c, it will become

$$Ch2(a \to STOP \sqcap b \to STOP, a \to STOP \sqcap b \to STOP) \tag{9.3}$$

Thereby allowing for both choices on a or b. Q however, will be either of the following

$$Ch2(a \to STOP, a \to STOP) \quad \text{or} \quad Ch2(b \to STOP, b \to STOP) \tag{9.4}$$

This will only allow one of a or b when rolled back to the checkpoint.

Some investigation is needed on this subject, to see if it is possible to define a mechanism that lets us checkpoint every type of processes.

9.3 Moving Processes After Checkpointing

When we checkpoint a process, we save the variables it is using. Having an identical process on a different machine, this process could use the same checkpoint, and start from the previous process's checkpoint. That is, we can save a checkpoint to file, move the checkpoint, e.g. to a different server, and run it from the checkpoint. This can be useful when we have processes, that you want to see if works, but you do not want them to finish on your PC.

This should be implemented into PyCSP, so that a process can choose to terminate after it has saved its variables to a file.

9.4 No side-effects

In section 6.3.5 I wrote that no side-effects are allowed between two checkpoints. This is never enforced in the implementation. An implementation, disallowing side-effects, or destroying checkpoints on side-effects, should be made.

Chapter 10

Conclusion

The basics of CSP channels has been discussed. Any-to-any channels have been constructed using the interleaving operator. It has been shown that the three other channel types, one-to-one, any-to-one, and one-to-any, can be made from the any-to-any channel. With or without the interleaving operator, buffered channels can still exist, with the help of a buffering process, which accepts all communication, and passes it on. Channels can be made with a choice operator as well, however this requires $n \cdot m$ communication events, where n is the amount of readers and m is the amount of writers.

With the help of a supervisor process, a process that overlooks the communications on a channel, poison has been formalised to work on any-to-any channels. The supervisor process can disallow communications on the channel, because it has the channel in its alphabet, and are required to do the communication events synchronised. Again, as the other three types of channels can be made from any-to-any channels, poison works on these as well.

Poisons less aggressive brother, retirement, has been discussed. With retirement a channel is closed on the last retirement instead of the first poison. That way we can avoid e.g. sending the number of jobs from a producer to a consumer, as the workers will not be shut down until there are no more work, and they will not propagate this shut down, until every worker is done.

Implicit retirement has been discussed as a way of helping programmers to not think about the shut down of the network. When a process terminates, it automatically retires all of its channels. Implicit retirement has been implemented in PyCSP as well as shown in the CSP algebra as wrapper function.

The supervisor paradigm has been used to introduce exception handling in the CSP algebra. Five different exception patterns has been discussed, fail-stop, retire-like fail-stop, broadcast, message replay and checkpointing. Fail-stop poisons the network, when a process has entered an exception state. Retire-like fail-stop only retires that process's channels. With broadcast, a message is sent to all subscribing processes, that this one has failed. Message replay rely on the messages being transformed into objects, having an id and a receiver. If a process fails, all messages sent to that process can be replayed, e.g. on an any-to-any channel, where other processes can pick up the work. Checkpointing saves the current state of a process after each communication event. Upon failure, this process and processes communicating with this one, are rolled back into the previous checkpoint. From here, the processes are restarted, given another chance to fulfil their jobs.

Fail-stop, retire-like fail-stop and checkpointing have been implemented in PyCSP. Each can be set on in the Process decorator. A number of retries for checkpointing can be set, as well as a different exception pattern, if this number is reached.

In addition to this thesis, a paper (appendix A) was submitted and accepted to Communicating Process Architectures 2012, a conference on concurrent and parallel programming.

Bibliography

[Bro07] Neil C. C. Brown. C++CSP2: A Many-to-Many Threading Model for Multicore Architectures. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 183–205, jul 2007.

[Bro08] Neil C.C. Brown. Communicating Haskell Processes: Composable Explicit Concurrency Using Monads. In Peter H. Welch, S. Stepney, F.A.C Polack, Frederick R. M. Barnes, Alistair A. McEwan, G. S. Stiles, Jan F. Broenink, and Adam T. Sampson, editors, *Communicating Process Architectures 2008*, pages 67–83, sep 2008.

[BVA07] John Markus Bjørndalen, Brian Vinter, and Otto J. Anshus. PyCSP - Communicating Sequential Processes for Python. In Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 229–248, jul 2007.

[BW03] 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.

[FVB10] Rune Møllegård Friborg, Brian Vinter, and John Markus Bjørndalen. Pycsp - controlled concurrency. *IJIPM*, 1(2):40–49, 2010.

[Hil05a] Gerald H. 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.

[Hil05b] Gerald Henk Hilderink. *Managing complexity of control software through concurrency*. PhD thesis, Enschede, May 2005.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[RHB97] 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.

[Ros10] A. W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.

[Ros11] A. W. Roscoe. On the expressiveness of CSP. feb 2011.

[SA05] Bernhard Sputh and Alastair R. Allen. JCSP-Poison: Safe Termination of CSP Process Networks. In *Communicating Process Architectures 2005*, pages 71–107, sep 2005.

[VBF09] Brian Vinter, John Markus Bjørndalen, and Rune Møllegård 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.

[WM00] Peter H. Welch and Jeremy M. R. Martin. Formal Analysis of Concurrent Java Systems. In Peter H. Welch and Andrè W. P. Bakkers, editors, *Communicating Process Architectures 2000*, pages 275–301, sep 2000.

Appendix A

Exception Handling and Checkpointing in CSP paper

Communicating Process Architectures 2012
P.H. Welch et al. (Eds.)
Draft, 2012
© *2012 The authors. All rights reserved.*

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 is 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 is 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 Larsens master 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 exists: 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 has 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.

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

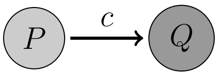


Figure 1. One-to-one channel

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.

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

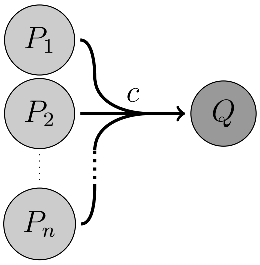


Figure 2. Any-to-one channel

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.

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

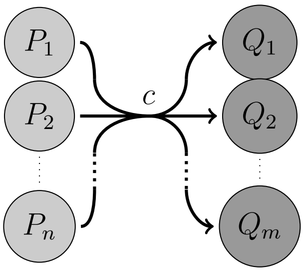


Figure 3. Any-to-any channel

Each step one of the P_i writers get to write to the channel and of the the Q_j readers get to read.

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

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

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:

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

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 respectively Q's channels. S_e is a process which will poison the processes that shares c. Figure 4 shows how these processes interact.

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

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

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

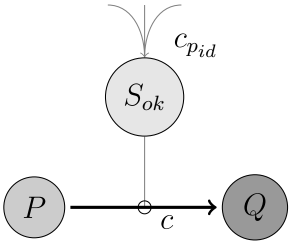


Figure 4. Poison on one-to-one channel

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.

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

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).

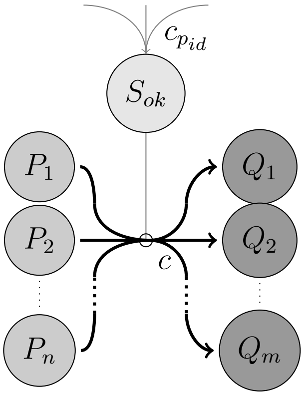


Figure 5. Poison on any-to-any channel

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:

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

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

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

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 $\overrightarrow{\Delta}$ 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:

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

Roscoe continues Hoares 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 Roscoes 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 exception, with which it can 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.

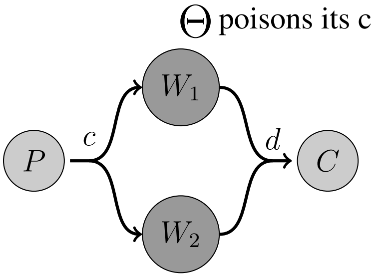


Figure 7. Fail-stop in worker process

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 works in the same way.

4.4.2. Retire-like Fail-stop

While fail-stop resembles poison, this pattern instead mimics retire. The information sent to the process 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 thing 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 that 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 thing that cannot be rolled back.

Checkpoints are quite similar to transactions, as we know them from SQL, in that we either do all the things 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 communicate 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 it's last checkpoint.If Process B has in fact terminated, Process A should enter an exception state, and possible 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 $\textcircled{c}$ [1] as well as a roll back event $\textcircled{r}$. With this, we can define a new process $Ch(P)$ which behaves like P, but also incorporates checkpoints. We assume that $\textcircled{c}, \textcircled{r} \notin \alpha P$. 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 $P = (x : A \rightarrow P(x))$, then $Ch2(P, Q)$ is defined as

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

That is, the process P is working as usual, but upon the event $\textcircled{c}$ we save the current P as our checkpoint. Upon $\textcircled{r}$ 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 $\textcircled{c}$ 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:

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

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

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

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 be two processes which sends each other a message, and forwards 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'.

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

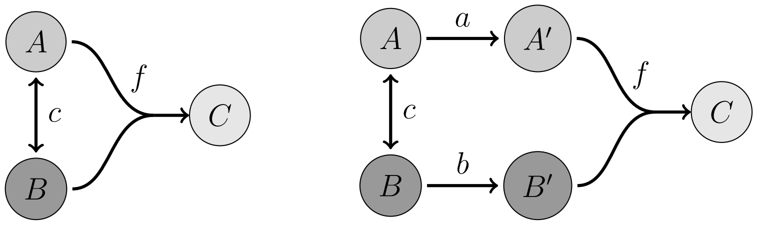
A supervisor is needed for each pair of communication events:

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

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 master 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 goes 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.

Appendix B

PyCSP code

B.1 `const.py`

```
"""
Constants

Copyright (c) 2009 John Markus Bjoerndalen <jmb@cs.uit.no>,
      Brian Vinter <vinter@diku.dk>, Rune M. Friborg <runef@diku.dk>.
See LICENSE.txt for licensing details (MIT License).
"""

# Operation type
READ, WRITE = range(2)

# Result of a channel request (ChannelReq)
FAIL, SUCCESS = range(2)

# State of a channel request status (ReqStatus)
ACTIVE, DONE = range(2)

# Constants used for both ChannelReq results and ReqStatus states.
NONE, POISON, RETIRE, FAILSTOP, RETIRELIKE, CHECKPOINT = range(1,7)

# Checkpoint retries
CHECKPOINT_RETRIES = 2
```

B.2 `__init__.py`

```
#!/usr/bin/env python
# -*- coding: latin-1 -*-
"""
PyCSP implementation of the CSP Core functionality (Channels, Processes, PAR, ALT).

Copyright (c) 2009 John Markus Bjoerndalen <jmb@cs.uit.no>,
      Brian Vinter <vinter@diku.dk>, Rune M. Friborg <runef@diku.dk>.
Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:
```

```
The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.  THE
SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
"""

# Imports
from guard import Skip, Timeout, SkipGuard, TimeoutGuard
from alternation import choice, Alternation
from altselect import FairSelect, AltSelect, InputGuard, OutputGuard
from channel import Channel, ChannelPoisonException, ChannelRetireException, ChannelFailstopEx
from channelend import retire, poison, IN, OUT
from process import io, Process, process, Sequence, Parallel, Spawn, current_process_id, load_

version = (0,7,1, 'threads')

# Set current implementation
import pycsp.current
pycsp.current.version = version
pycsp.current.trace = False

pycsp.current.Skip = Skip
pycsp.current.Timeout = Timeout
pycsp.current.SkipGuard = SkipGuard
pycsp.current.TimeoutGuard = TimeoutGuard
pycsp.current.choice = choice
pycsp.current.Alternation = Alternation
pycsp.current.Channel = Channel
pycsp.current.ChannelPoisonException = ChannelPoisonException
pycsp.current.ChannelRetireException = ChannelRetireException
pycsp.current.ChannelFailstopException = ChannelFailstopException
pycsp.current.ChannelRetireLikeFailstopException = ChannelRetireLikeFailstopException
pycsp.current.ChannelRollBackException = ChannelRollBackException
pycsp.current.retire = retire
pycsp.current.poison = poison
pycsp.current.IN = IN
pycsp.current.OUT = OUT
pycsp.current.io = io
pycsp.current.Process = Process
pycsp.current.process = process
pycsp.current.Sequence = Sequence
pycsp.current.Parallel = Parallel
pycsp.current.Spawn = Spawn
pycsp.current.current_process_id = current_process_id
pycsp.current.FairSelect = FairSelect
pycsp.current.AltSelect = AltSelect
pycsp.current.InputGuard = InputGuard
pycsp.current.OutputGuard = OutputGuard
pycsp.current.load_variables = load_variables
pycsp.current.load = load

def test_suite():
    import unittest
    import doctest
    import alternation, channel, channelend, process, guard, buffer

```

```
    suite = unittest.TestSuite()
    for mod in alternation, channel, channelend, process, guard, buffer:
        suite.addTest(doctest.DocTestSuite(mod))
    suite.addTest(doctest.DocTestSuite())
    return suite
```

B.3 channel.py

```
"""
Channel module

Copyright (c) 2009 John Markus Bjoerndalen <jmb@cs.uit.no>,
      Brian Vinter <vinter@diku.dk>, Rune M. Friborg <runef@diku.dk>.
Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.  THE
SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
"""
# Imports
import threading
import inspect
import time, random
from channelend import ChannelRetireException, ChannelRetireLikeFailstopException, ChannelEnd
from pycsp.common.const import *

# Exceptions
class ChannelPoisonException(Exception):
    def __init__(self):
        pass

class ChannelFailstopException(Exception):
    def __init__(self):
        pass

class ChannelRollBackException(Exception):
    def __init__(self):
        pass

# Classes
class ReqStatus:
    def __init__(self, state=ACTIVE):
        self.state=state
        self.cond = threading.Condition()

class ChannelReq:
    def __init__(self, status, msg=None, signal=None, name=None):
```

```
        self.status=status
        self.msg=msg
        self.signal=signal
        self.result=FAIL
        self.name=name

    def cancel(self):
        self.status.cond.acquire()
        self.status.state=CANCEL
        self.status.cond.notifyAll()
        self.status.cond.release()

    def poison(self):
        self.status.cond.acquire()
        if self.result == FAIL and self.status.state == ACTIVE:
            self.status.state=POISON
            self.result=POISON
            self.status.cond.notifyAll()
        self.status.cond.release()

    def retire(self):
        self.status.cond.acquire()
        if self.result == FAIL and self.status.state == ACTIVE:
            self.status.state=RETIRE
            self.result=RETIRE
            self.status.cond.notifyAll()
        self.status.cond.release()

    def failstop(self):
        self.status.cond.acquire()
        if self.result == FAIL and self.status.state == ACTIVE:
            self.status.state=FAILSTOP
            self.result=FAILSTOP
            self.status.cond.notifyAll()
        self.status.cond.release()

    def retirelike(self):
        self.status.cond.acquire()
        if self.result == FAIL and self.status.state == ACTIVE:
            self.status.state=RETIRELIKE
            self.result=RETIRELIKE
            self.status.cond.notifyAll()
        self.status.cond.release()

    def wait(self):
        self.status.cond.acquire()
        while self.status.state==ACTIVE:
            self.status.cond.wait()
        self.status.cond.release()

    def offer(self, recipient):
        # Eliminate unnecessary locking, by adding an extra test
        if self.status.state == recipient.status.state == ACTIVE:

            s_cond = self.status.cond
            r_cond = recipient.status.cond

            # Ensuring to lock in the correct order.
            if s_cond < r_cond:
                s_cond.acquire()
                r_cond.acquire()
```

```
            else:
                r_cond.acquire()
                s_cond.acquire()

            if self.status.state == recipient.status.state == ACTIVE:
                recipient.msg=self.msg
                self.status.state=DONE
                self.result=SUCCESS
                recipient.status.state=DONE
                recipient.result=SUCCESS
                s_cond.notifyAll()
                r_cond.notifyAll()

            # Ensuring that we also release in the correct order. ( done in the opposite orde
            if s_cond < r_cond:
                r_cond.release()
                s_cond.release()
            else:
                s_cond.release()
                r_cond.release()



class Channel(object):
    """ Channel class. Blocking communication

    >>> from __init__ import *

    >>> @process
    ... def P1(cout):
    ...     while True:
    ...         cout('Hello World')

    >>> C = Channel()
    >>> Spawn(P1(C.writer()))

    >>> cin = C.reader()
    >>> cin()
    'Hello World'

    >>> retire(cin)
    """
    def __new__(cls, *args, **kargs):
        if kargs.has_key('buffer') and kargs['buffer'] > 0:
            import buffer
            chan = buffer.BufferedChannel(*args, **kargs)
            return chan
        else:
            return object.__new__(cls)

    def __init__(self, name=None, buffer=0):
        self.readqueue  = []
        self.writequeue = []

        self.status = NONE
        self.old_status = NONE

        self.readers = 0
        self.writers = 0

        if name == None:
```

```
            # Create unique name
            self.name = str(random.random())+str(time.time())
        else:
            self.name=name

        # This lock is used to ensure atomic updates of the channelend
        # reference counting and to protect the read/write queue operations.
        self.lock = threading.RLock()

    def save_variables(self):
        stack = inspect.stack()

        try:
            locals_ = stack[2][0].f_locals
            process_ = stack[3][0].f_locals
        finally:
            del stack

        process_['self'].vars = locals_

    def check_termination(self):
        if self.status == POISON:
            raise ChannelPoisonException()
        elif self.status == RETIRE:
            raise ChannelRetireException()
        elif self.status == FAILSTOP:
            raise ChannelFailstopException()
        elif self.status == RETIRELIKE:
            raise ChannelRetireLikeFailstopException()
        elif self.status == CHECKPOINT:
            self.status = self.old_status
            raise ChannelRollBackException()

    def _read(self):
        self.check_termination()
        req=ChannelReq(ReqStatus(), name=self.name)
        self.post_read(req)
        req.wait()
        self.remove_read(req)
        if req.result==SUCCESS:
            self.save_variables()
            return req.msg
        self.check_termination()

        print 'We should not get here in read!!!', req.status.state
        return None

    def _write(self, msg):
        self.check_termination()
        req=ChannelReq(ReqStatus(), msg)
        self.post_write(req)
        req.wait()
        self.remove_write(req)
        if req.result==SUCCESS:
            self.save_variables()
            return
        self.check_termination()

        print 'We should not get here in write!!!', req.status
        return
```

```
    def post_read(self, req):
        self.check_termination()

        success = True
        self.lock.acquire()
        if self.status != NONE:
            success = False
        else:
            self.readqueue.append(req)
        self.lock.release()

        if success:
            self.match()
        else:
            self.check_termination()

    def remove_read(self, req):
        self.lock.acquire()
        self.readqueue.remove(req)
        self.lock.release()

    def post_write(self, req):
        self.check_termination()

        success = True
        self.lock.acquire()
        if self.status != NONE:
            success = False
        else:
            self.writequeue.append(req)
        self.lock.release()

        if success:
            self.match()
        else:
            self.check_termination()

    def remove_write(self, req):
        self.lock.acquire()
        self.writequeue.remove(req)
        self.lock.release()

    def match(self):
        self.lock.acquire()
        for w in self.writequeue:
            for r in self.readqueue:
                w.offer(r)
        self.lock.release()

    def poison(self):
        self.lock.acquire()
        self.status=POISON
        for p in self.readqueue:
            p.poison()
        for p in self.writequeue:
            p.poison()
        self.lock.release()

    def failstop(self):
        self.lock.acquire()
        self.status=FAILSTOP
```

```
        for p in self.readqueue:
            p.failstop()
        for p in self.writequeue:
            p.failstop()
        self.lock.release()

    def rollback(self):
        self.lock.acquire()

        if self.status != CHECKPOINT:
            self.old_status = self.status
            self.status = CHECKPOINT

        self.lock.release()

    # syntactic sugar: cin = +chan
    def __pos__(self):
        return self.reader()

    # syntactic sugar: cout = -chan
    def __neg__(self):
        return self.writer()

    # syntactic sugar: Channel() * N
    def __mul__(self, multiplier):
        new = [self]
        for i in range(multiplier-1):
            new.append(Channel(name=self.name+str(i+1)))
        return new

    # syntactic sugar: N * Channel()
    def __rmul__(self, multiplier):
        return self.__mul__(multiplier)

    def reader(self):
        """
        Join as reader

        >>> C = Channel()
        >>> cin = C.reader()
        >>> isinstance(cin, ChannelEndRead)
        True
        """
        self.join_reader()
        return ChannelEndRead(self)

    def writer(self):
        """
        Join as writer

        >>> C = Channel()
        >>> cout = C.writer()
        >>> isinstance(cout, ChannelEndWrite)
        True
        """

        self.join_writer()
        return ChannelEndWrite(self)

    def join_reader(self):
        self.lock.acquire()
```

```
            self.readers+=1
            self.lock.release()

    def join_writer(self):
            self.lock.acquire()
            self.writers+=1
            self.lock.release()

    def leave_reader(self, status=RETIRE):
            self.lock.acquire()
            if self.status != RETIRE or self.status != RETIRELIKE:
                self.readers-=1
                if self.readers==0:
                    # Set channel retired
                    self.status = status
                    for p in self.writequeue:
                        if status == RETIRELIKE:
                            p.retirelike()
                        else:
                            p.retire()
            self.lock.release()

    def leave_writer(self, status=RETIRE):
            self.lock.acquire()
            if self.status != RETIRE or self.status != RETIRELIKE:
                self.writers-=1
                if self.writers==0:
                    # Set channel retired
                    self.status = status
                    for p in self.readqueue:
                        if status == RETIRELIKE:
                            p.retirelike()
                        else:
                            p.retire()
            self.lock.release()

# Run tests
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

B.4 channelend.py

```
"""
Channelend module

Copyright (c) 2009 John Markus Bjoerndalen <jmb@cs.uit.no>,
      Brian Vinter <vinter@diku.dk>, Rune M. Friborg <runef@diku.dk>.
Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.  THE
SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
```

```
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
"""
from pycsp.common.const import *

# Exceptions
class ChannelRetireException(Exception):
    def __init__(self):
        pass

class ChannelRetireLikeFailstopException(Exception):
    def __init__(self):
        pass

# Functions
def IN(channel):
    """ Join as reader
    """
    print ’Warning: IN() are deprecated and will be removed’
    return channel.reader()

def OUT(channel):
    """ Join as writer
    """
    print ’Warning: OUT() are deprecated and will be removed’
    return channel.writer()

def retire(*list_of_channelEnds):
    """ Retire reader or writer, to do auto-poisoning
    When all readers or writer of a channel have retired. The channel is retired.

    >>> from __init__ import *
    >>> C = Channel()
    >>> cout1, cout2 = C.writer(), C.writer()
    >>> retire(cout1)

    >>> Spawn(Process(cout2, ’ok’))

    >>> try:
    ...     cout1(’fail’)
    ... except ChannelRetireException:
    ...     True
    True

    >>> cin = C.reader()
    >>> retire(cin)
    """
    for channelEnd in list_of_channelEnds:
        channelEnd.retire()

def poison(*list_of_channelEnds):
    """ Poison channel
    >>> from __init__ import *

    >>> @process
    ... def P1(cin, done):
    ...     try:
```

```
    ...   while True:
    ...     cin()
    ... except ChannelPoisonException:
    ...   done(42)

    >>> C1, C2 = Channel(), Channel()
    >>> Spawn(P1(C1.reader(), C2.writer()))
    >>> cout = C1.writer()
    >>> cout('Test')

    >>> poison(cout)

    >>> cin = C2.reader()
    >>> cin()
    42
    """
    for channelEnd in list_of_channelEnds:
        channelEnd.poison()

def failstop(*list_of_channelEnds):
    for channelEnd in list_of_channelEnds:
        channelEnd.failstop()

def retirelike(*list_of_channelEnds):
    for channelEnd in list_of_channelEnds:
        channelEnd.retirelike()

# Classes
class ChannelEndWrite:
    def __init__(self, channel):
        self.channel = channel
        self.op = WRITE

        # Prevention against multiple retires
        self.isretired = False

        self.__call__ = self.channel._write
        self.post_write = self.channel.post_write
        self.remove_write = self.channel.remove_write
        self.poison = self.channel.poison
        self.failstop = self.channel.failstop
        self.rollback = self.channel.rollback

    def _retire(self, *ignore):
        raise ChannelRetireException()

    def _retirelike(self, *ignore):
        raise ChannelRetireLikeFailstopException()

    def retire(self):
        if not self.isretired and self.channel.status != POISON and self.channel.status != FA
            self.channel.leave_writer()
            self.__call__ = self._retire
            self.post_write = self._retire
            self.isretired = True

    def retirelike(self):
        if not self.isretired and self.channel.status != POISON and self.channel.status != FA
            self.channel.leave_writer(RETIRELIKE)
            self.__call__ = self._retirelike
            self.post_write = self._retirelike
```

```
            self.isretired = True

    def __repr__(self):
        if self.channel.name == None:
            return "<ChannelEndWrite wrapping %s>" % self.channel
        else:
            return "<ChannelEndWrite wrapping %s named %s>" % (self.channel, self.channel.name

    def isWriter(self):
        return True

    def isReader(self):
        return False

class ChannelEndRead:
    def __init__(self, channel):
        self.channel = channel
        self.op = READ

        # Prevention against multiple retires
        self.isretired = False

        self.__call__ = self.channel._read
        self.post_read = self.channel.post_read
        self.remove_read = self.channel.remove_read
        self.poison = self.channel.poison
        self.failstop = self.channel.failstop
        self.rollback = self.channel.rollback

    def _retire(self, *ignore):
        raise ChannelRetireException()

    def _retirelike(self, *ignore):
        raise ChannelRetireLikeFailstopException()

    def retire(self):
        if not self.isretired and self.channel.status != POISON and self.channel.status != FAI
            self.channel.leave_reader()
            self.__call__ = self._retire
            self.post_read = self._retire
            self.isretired = True

    def retirelike(self):
        if not self.isretired and self.channel.status != POISON and self.channel.status != FAI
            self.channel.leave_reader(RETIRELIKE)
            self.__call__ = self._retirelike
            self.post_read = self._retirelike
            self.isretired = True

    def __repr__(self):
        if self.channel.name == None:
            return "<ChannelEndRead wrapping %s>" % self.channel
        else:
            return "<ChannelEndRead wrapping %s named %s>" % (self.channel, self.channel.name)

    def isWriter(self):
        return False

    def isReader(self):
        return True
```

```
# Run tests
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

B.5 process.py

```
"""
Processes and execution

Copyright (c) 2009 John Markus Bjoerndalen <jmb@cs.uit.no>,
      Brian Vinter <vinter@diku.dk>, Rune M. Friborg <runef@diku.dk>.
Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.  THE
SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
"""

# Imports
import inspect, sys
import types
import threading
import time, random
from channel import ChannelPoisonException, ChannelRetireException, ChannelFailstopException,
from channelend import ChannelEndRead, ChannelEndWrite
from pycsp.common.const import *

# Decorators
def process(func=None, **options):
    """
    @process decorator for creating process functions

    >>> @process
    ... def P():
    ...     pass

    >>> isinstance(P(), Process)
    True

    Processes can have a fail_type.
    This is checked when failing.

    >>> @process(fail_type=FAILSTOP)
    ... def P():
    ...     1/0
    """
```

```
    if func != None:
        def _call(*args, **kwargs):
            return Process(func, options, *args, **kwargs)
        return _call
    else:
        def _func(func):
            return process(func, **options)
        return _func

def io(func):
    """
    @io decorator for blocking io operations.
    In PyCSP threading it has no effect, other than compatibility

    >>> @io
    ... def sleep(n):
    ...     import time
    ...     time.sleep(n)

    >>> sleep(0.01)
    """
    return func

def load_variables(*pargs):
    stack = inspect.stack()

    try:
        process_ = stack[3][0].f_locals
    finally:
        del stack

    loaded_vars = process_['self'].vars

    var = []
    for __x in pargs:
        if __x[0] in loaded_vars:
            var.append(loaded_vars[__x[0]])
        else:
            var.append(__x[1])

    if len(var) == 1:
        return var[0]
    else:
        return var

def load(**kwargs):
    if len(kwargs) > 1:
        raise AttributeError

    for __x, __v in kwargs.iteritems():
        return load_variables((__x, __v))

# Classes
class Process(threading.Thread):
    """ Process(func, *args, **kwargs)
    It is recommended to use the @process decorator, to create Process instances
    """
    def __init__(self, fn, options, *args, **kwargs):
        threading.Thread.__init__(self)
        self.fn = fn
```

```
        self.fail_type = None
        if options is not None and ’fail_type’ in options:
            self.fail_type = options[’fail_type’]

        self.args = args
        self.kwargs = kwargs

        # Create unique id
        self.id = str(random.random())+str(time.time())

        self.options = options
        self.vars = {}

        self.print_error = False
        if options is not None and ’print_error’ in options:
            self.print_error = options[’print_error’]

        self.max_retries = CHECKPOINT_RETRIES
        if options is not None and ’retries’ in options:
            self.max_retries = options[’retries’]

        self.retries = 0

        self.fail_type_after_retries = self.__check_retirelike
        if options is not None and ’fail_type_after_retries’ in options:
            if options[’fail_type_after_retries’] == FAILSTOP:
                self.fail_type_after_retries = self.__check_failstop

    def run(self):
        try:
            # Store the returned value from the process
            self.fn(*self.args, **self.kwargs)
            # The process is done
            # It should auto retire all of its channels
            self.__check_retire(self.args)
            self.__check_retire(self.kwargs.values())
        except ChannelPoisonException:
            # look for channels and channel ends
            self.__check_poison(self.args)
            self.__check_poison(self.kwargs.values())
        except ChannelRetireException:
            # look for channel ends
            self.__check_retire(self.args)
            self.__check_retire(self.kwargs.values())
        except ChannelFailstopException:
            self.__check_failstop(self.args)
            self.__check_failstop(self.kwargs.values())
        except ChannelRetireLikeFailstopException:
            self.__check_retirelike(self.args)
            self.__check_retirelike(self.kwargs.values())
        except ChannelRollBackException:
            # Another process sharing a channel with this one
            # has rolled back, so we must as well.
            self.run()
        except Exception as e:
            if self.print_error:
                print e

            fail_type_fn = None
            rerun = False

```

```
            if self.fail_type == FAILSTOP:
                fail_type_fn = self.__check_failstop
            elif self.fail_type == RETIRELIKE:
                fail_type_fn = self.__check_retirelike
            elif self.fail_type == CHECKPOINT:
                if self.max_retries != -1 and self.retries >= self.max_retries:
                    fail_type_fn = self.fail_type_after_retries
                else:
                    rerun = True
                    fail_type_fn = self.__check_checkpointing

            if fail_type_fn is not None:
                fail_type_fn(self.args)
                fail_type_fn(self.kwargs.values())

            if rerun:
                self.retries += 1
                self.run()

    def __check_poison(self, args):
        for arg in args:
            try:
                if types.ListType == type(arg) or types.TupleType == type(arg):
                    self.__check_poison(arg)
                elif types.DictType == type(arg):
                    self.__check_poison(arg.keys())
                    self.__check_poison(arg.values())
                elif type(arg.poison) == types.UnboundMethodType:
                    arg.poison()
            except AttributeError:
                pass

    def __check_retire(self, args):
        for arg in args:
            try:
                if types.ListType == type(arg) or types.TupleType == type(arg):
                    self.__check_retire(arg)
                elif types.DictType == type(arg):
                    self.__check_retire(arg.keys())
                    self.__check_retire(arg.values())
                elif type(arg.retire) == types.UnboundMethodType:
                    # Ignore if try to retire an already retired channel end.
                    try:
                        arg.retire()
                    except ChannelRetireException:
                        pass
                    except ChannelRetireLikeFailstopException:
                        pass
            except AttributeError:
                pass

    def __check_failstop(self, args):
        for arg in args:
            try:
                if types.ListType == type(arg) or types.TupleType == type(arg):
                    self.__check_failstop(arg)
                elif types.DictType == type(arg):
                    self.__check_failstop(arg.keys())
                    self.__check_failstop(arg.values())
                elif type(arg.failstop) == types.UnboundMethodType:
                    arg.failstop()
```

```python
            except AttributeError:
                pass

    def __check_retirelike(self, args):
        for arg in args:
            try:
                if types.ListType == type(arg) or types.TupleType == type(arg):
                    self.__check_retirelike(arg)
                elif types.DictType == type(arg):
                    self.__check_retirelike(arg.keys())
                    self.__check_retirelike(arg.values())
                elif type(arg.retirelike) == types.UnboundMethodType:
                    # Ignore if try to retire an already retired channel end.
                    try:
                        arg.retirelike()
                    except ChannelRetireLikeFailstopException:
                        pass
                    except ChannelRetireException:
                        pass
            except AttributeError:
                pass

    def __check_checkpointing(self, args):
        for arg in args:
            try:
                if types.ListType == type(arg) or types.TupleType == type(arg):
                    self.__check_checkpointing(arg)
                elif types.DictType == type(arg):
                    self.__check_checkpointing(arg.keys())
                    self.__check_checkpointing(arg.values())
                elif type(arg.rollback) == types.UnboundMethodType:
                    # Our argument is a channel
                    arg.rollback()
            except AttributeError:
                pass

    # syntactic sugar:  Process() * 2 == [Process<1>,Process<2>]
    def __mul__(self, multiplier):
        return [self] + [Process(self.fn, self.options, *self.__mul_channel_ends(self.args), *

    # syntactic sugar:  2 * Process() == [Process<1>,Process<2>]
    def __rmul__(self, multiplier):
        return self.__mul__(multiplier)

    # Copy lists and dictionaries
    def __mul_channel_ends(self, args):
        if types.ListType == type(args) or types.TupleType == type(args):
            R = []
            for item in args:
                try:
                    if type(item.isReader) == types.UnboundMethodType and item.isReader():
                        R.append(item.channel.reader())
                    elif type(item.isWriter) == types.UnboundMethodType and item.isWriter():
                        R.append(item.channel.writer())
                except AttributeError:
                    if item == types.ListType or item == types.DictType or item == types.Tuple
                        R.append(self.__mul_channel_ends(item))
                    else:
                        R.append(item)

            if types.TupleType == type(args):
```

```
                return tuple(R)
            else:
                return R

        elif types.DictType == type(args):
            R = {}
            for key in args:
                try:
                    if type(key.isReader) == types.UnboundMethodType and key.isReader():
                        R[key.channel.reader()] = args[key]
                    elif type(key.isWriter) == types.UnboundMethodType and key.isWriter():
                        R[key.channel.writer()] = args[key]
                    elif type(args[key].isReader) == types.UnboundMethodType and args[key].is
                        R[key] = args[key].channel.reader()
                    elif type(args[key].isWriter) == types.UnboundMethodType and args[key].is
                        R[key] = args[key].channel.writer()
                except AttributeError:
                    if args[key] == types.ListType or args[key] == types.DictType or args[key
                        R[key] = self.__mul_channel_ends(args[key])
                    else:
                        R[key] = args[key]
            return R
        return args

# Functions
def Parallel(*plist):
    """ Parallel(P1, [P2, .. ,PN])
    >>> from __init__ import *

    >>> @process
    ... def P1(cout, id):
    ...     for i in range(10):
    ...         cout(id)

    >>> @process
    ... def P2(cin):
    ...     for i in range(10):
    ...         cin()

    >>> C = [Channel() for i in range(10)]
    >>> Cin = [chan.reader() for chan in C]
    >>> Cout = [chan.writer() for chan in C]

    >>> Parallel([P1(Cout[i], i) for i in range(10)],[P2(Cin[i]) for i in range(10)])
    """
    _parallel(plist, True)

def Spawn(*plist):
    """ Spawn(P1, [P2, .. ,PN])
    >>> from __init__ import *

    >>> @process
    ... def P1(cout, id):
    ...     for i in range(10):
    ...         cout(id)

    >>> C = Channel()
    >>> Spawn([P1(C.writer(), i) for i in range(10)])

    >>> L = []
    >>> cin = C.reader()
```

```
    >>> for i in range(100):
    ...    L.append(cin())

    >>> len(L)
    100
    """
    _parallel(plist, False)

def _parallel(plist, block = True):
    processes=[]
    for p in plist:
        if type(p)==list:
            for q in p:
                processes.append(q)
        else:
            processes.append(p)

    for p in processes:
        p.start()

    if block:
        for p in processes:
            p.join()


def Sequence(*plist):
    """ Sequence(P1, [P2, .. ,PN])
    The Sequence construct returns when all given processes exit.
    >>> from __init__ import *

    >>> @process
    ... def P1(cout):
    ...     Sequence([Process(cout,i) for i in range(10)])

    >>> C = Channel()
    >>> Spawn(P1(C.writer()))

    >>> L = []
    >>> cin = C.reader()
    >>> for i in range(10):
    ...    L.append(cin())

    >>> L
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    """
    processes=[]
    for p in plist:
        if type(p)==list:
            for q in p:
                processes.append(q)
        else:
            processes.append(p)

    # For every process we simulate a new process_id. When executing
    # in Main thread/process we set the new id in a global variable.
    try:
        # compatible with Python 2.6+
        t = threading.current_thread()
        name = t.name
    except AttributeError:
        # compatible with Python 2.5-
```

```
        t = threading.currentThread()
        name = t.getName()

    if name == 'MainThread':
        global MAINTHREAD_ID
        for p in processes:
            MAINTHREAD_ID = p.id

            # Call Run directly instead of start() and join()
            p.run()
        del MAINTHREAD_ID
    else:
        t_original_id = t.id
        for p in processes:
            t.id = p.id

            # Call Run directly instead of start() and join()
            p.run()
        t.id = t_original_id

def current_process_id():
    try:
        # compatible with Python 2.6+
        t = threading.current_thread()
        name = t.name
    except AttributeError:
        # compatible with Python 2.5-
        t = threading.currentThread()
        name = t.getName()

    if name == 'MainThread':
        try:
            return MAINTHREAD_ID
        except NameError:
            return '__main__'
    return t.id

# Run tests
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```