Communicating Process Architectures 2005
Jan Broenink, Herman Roebbers, Johan Sunter, Peter Welch, and David Wood (Eds.)
IOS Press, 2005

JCSP-Poison: Safe Termination of CSP Process Networks

Bernhard H.C. SPUTH and Alastair R. ALLEN

Department of Engineering, University of Aberdeen, Aberdeen AB24 3UE, UK

bernhard@erg.abdn.ac.uk , a.allen@abdn.ac.uk

Abstract. This paper presents a novel technique for safe partial or complete process network termination. The idea is to have two types of termination messages / poison: LocalPoison and GlobalPoison. Injecting GlobalPoison into a process network results in a safe termination of the whole process network. In contrast, injected LocalPoison only terminates all processes until it is filtered out by Poison-Filtering Channels. This allows the creation of termination domains inside a process network. To make handling of a termination message easy, it is delivered as an exception and not as a normal message. The necessary Poisonable- and Poison-Filtering-Channels have been modelled in CSP and checked using FDR. A proof of concept implementation for Communicating Sequential Processes for Java (JCSP) has been developed and refined. Previously, JCSP offered no safe way to terminate the process network. When the user terminated the program, the Java Virtual Machine (JVM) simply stops all threads (processes), without giving the processes the chance to perform clean up operations. A similar technique is used to perform partial termination of process networks in JCSP, making it unsafe as well. The technique presented in this paper is not limited to JCSP, but can easily be ported to other CSP environments. Partial process network termination can be applied in the area of Software Defined Radio (SDR), because SDR systems need to be able to change their signal processing algorithms during runtime.

Keywords. JCSP, SDR, Partial Process Network Termination, Poisoning, Poionable-Channel, Poison-Filtering-Channel, Termination Domains

Introduction

In CSP [1,2] applications consist of processes. A process is a sequence of instructions. In complex applications multiple processes execute concurrently. To avoid race conditions when accessing global resources, processes are not allowed to share global resources without synchronisation. Processes communicate with each other by using unidirectional channels. A communication over a channel only takes place when receiver and sender processes are cooperating. This rendezvous of processes is used for synchronisation in CSP. The combination of processes and channels forms a *process network* .

Process networks can be visualised in the form of block diagrams. Processes are represented by blocks. Channels are represented by arrows, with the arrow head indicating the direction of communication of the channel. Figure 1 shows a simple process network, where the *PRODUCER* process sends messages, over a channel with name *messenger*, to the *CONSUMER* process.

To terminate a process network all processes of it need to terminate. In the example given this means the both *PRODUCER* and *CONSUMER* have to terminate. In CSP a process only terminates once it has fulfilled its task. A problem occurs when a process does not know when it has fulfilled its task. This is, for instance, the case for processes that are part of a signal processing chain. A signal processing chain consists of three parts: data source, signal

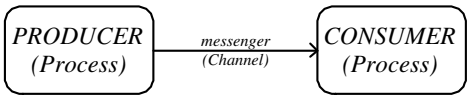


Figure 1. Simple process network diagram

processing part and a data sink, see figure 2. In such a chain, only the data source is able to determine that all signal processing has been performed. And this only if the data processing is done off-line, meaning that the signal data was provided in the form of a file. If the signal processing is performed online not even the data source knows when it has reached its end of usefulness. This decision is made by the user, who then has to close the application to stop the signal processing. But even if the data source is able to determine that the task is fulfilled, it has no way of telling the other processes of the chain. This is caused by the way signal processing processes operate, it is a constant loop of: inputting a frame of signal data, processing the frame and then outputting it. In the case of no signal data arriving anymore, these processes will wait infinitely for new signal data and thus never terminate. The following text will discuss different methods to enable safe termination of process networks. Furthermore, in Software Defined Radios (SDR)[3,4], it is necessary to exchange software modules, without affecting their execution environment, thus partial process network termination is necessary and a technique for it will be developed and discussed as well in this paper. Due to previous work by us in the field of signal processing and CSP done using JCSP [5], the implementation of the techniques shown in this paper are based upon JCSP.

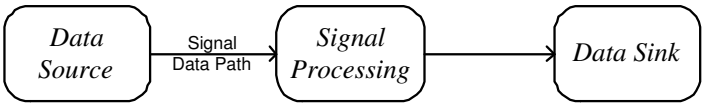


Figure 2. Principle Signal Processing Chain

1. Terminating Networks in JCSP

In this section the various available techniques of process network termination in JCSP [6] are introduced. The shortcomings of these techniques are discussed as well.

1.1. Process Network Termination in JCSP

JCSP [6] is an environment allowing the development of Java applications following CSP principles. In JCSP, processes are represented by Java threads. The Java Virtual Machine (JVM) differentiates between two types of threads, user threads and daemon threads. The definition of a daemon thread, according to [7, Page 26], is: ”A daemon is simply a thread that has no other role in life than to serve others”, with others meaning other threads. By contrast, a user thread serves the user of the program. For the JVM, a program which consists only of daemon threads does not perform any purpose and is terminated. As this can lead to unwanted program termination, all threads are created as user threads by default and can be converted to daemons afterwards. JCSP creates all threads as daemons. Only the thread created by the JVM itself, when starting the program, is a user thread. Termination of a complete JCSP process network can therefore be done by terminating the thread created by the JVM. This type of process network termination is simple to use but in most cases utterly unsafe. This is due to the fact that individual processes cannot perform clean up operations prior to their termination.

1.1.1. Partial Process Network Termination in JCSP

In JCSP it is possible to spin off process networks using the ProcessManager class. This class also provides the ability to stop the process network, that it spun off. According to the documentation this function should not be used. JCSP therefore, offers no special support for network termination. Instead the user must arrange for network termination himself.

1.1.2. Possible Solution

One possible way to terminate a process network is by broadcasting the termination message to all processes, of a process network. Upon reception of this message, the processes terminate. This can be achieved in JCSP by sharing a message variable among the group of processes to be terminated. There are two types of processes: one broadcaster process and multiple receiver processes. Access to this variable is synchronised on a JCSP Barrier shared among all processes. The JCSP Barrier splits each processing cycle into two phases, to comply with the CREW (Concurrent-Read, Exclusive-Write) concept:

- In the first phase all receiver processes check the shared message variable. Broadcaster process will not change the message variable during that time.
- In the second phase the broadcaster may change the shared message variable. The receiver processes must not look at the shared message variable.

Once the broadcaster decides to terminate the system, it changes the message variable and all processes, including the broadcaster, will see the termination message the next time they check the variable. This way all processes terminate in the same cycle. This approach is having a number of drawbacks:

- Broadcaster and receiver processes need to synchronise twice per cycle on the JCSP Barrier. This is an computing intensive task.
- It will only work for systems where all processes have the same cycle length. In signal processing this is not necessary the case. There a combiner process may require multiple signal data frames, from the process up the stream, to construct a single frame which is required by the later stages of processing. For the process upstream a single cycle is the creation of one frame, for the combiner process the cycle length is as well the creation of one frame, but it requires multiple frames from the upstream process to do so. If now the upstream process tries to synchronise with the JCSP Barrier it will have to wait for the combiner process as well to synchronise. But with the combiner process not having finished its cycle, it will not synchronise, instead it will wait for a frame to arrive from the process upstream. The result is a deadlock. Therefore, if in such a system the previous mentioned technique for broadcasting is used, the developer has to make sure that all processes have the same cycle length. In the example given, the upstream process would have to produce multiple output frames during one cycle.

1.2. Introduction to Graceful Termination

Not being able to perform clean up operations may lead to data corruption, or unnecessary resource consumption. In [8] P.H. Welch discusses different ways to perform safe termination of process networks, and introduces a technique called *graceful termination*. In this technique a special message, the poison, is sent to one process of the network, this is the reason why this technique is also referred to as *poisoning*.

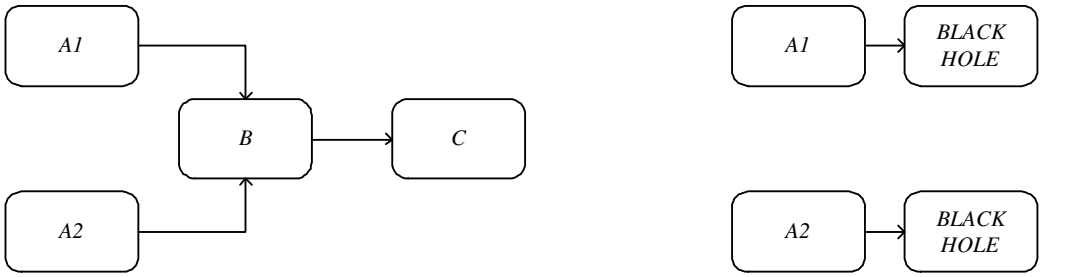
A process receiving poison:

1. Performs all necessary clean up operations;
2. Places *while-no-poison-black-hole-messages* processes, also known as *black-hole* processes at all its channel inputs. Such processes swallow any arriving messages, but terminate when poison is received.
3. Sends poison over all its channel outputs, to poison all processes it sends data to.
4. Does items 2. and 3. above in *parallel* and waits for all installed processes an the poison-sends to terminate. Then it terminates.

While this technique looks good at first glance there are several problems when trying to use it in practice. Why this is the case will be discussed in the following.

1.3. JCSP and Graceful Termination

The implementation of graceful termination as proposed in [8], relies on the ability of processes to differentiate between incoming normal messages and poison. But how to differentiate between a poisonous and a normal message? For the Object-channels available in JCSP, it can be easily done by defining a class representing poison, so the receiver of a message has only to perform a type checking operation. For integer-channels, this is not possible, as only integer values are passed. One possible solution is to send two integer values for a normal message. The first integer value indicating the poison or not (for instance use 0 for non poisoned and 1 for poisoned), the second the value to be transferred. In case of poison only the first integer is transmitted. This will work, but requires the double amount of bandwidth and increases the processor drain during normal operation. As this is undesirable, a different delivery mechanism for poison has to be found.



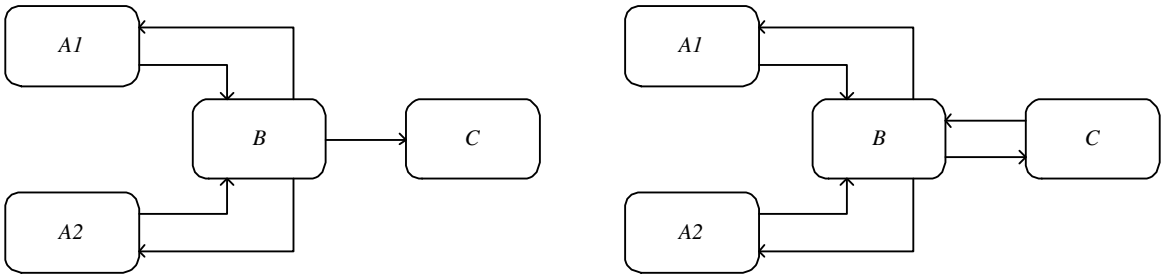
(a) Prior to termination (b) After termination of process B

Figure 3. Complex process network before and after termination of process B

A terminated process does not any longer fetch messages from its channel-inputs. If now processes try to send messages over the associated channels, these processes will wait indefinitely for the receiving process to fetch the message: these processes deadlock. To avoid this situation, the terminating process needs to somehow service its channel inputs. This is done by connecting each channel input with a process that simply swallows any incoming message, these processes are called *black hole* processes. This effectively prevent the process network from deadlocking.

If in the process network shown in Figure 3(a), process *B* terminates, it creates two *Black Hole* processes for its two channel inputs and sends poison to its channel output. Process *C* terminates, without creating a *Black Hole* process, due to it only being connected to one channel over which it received poison. As illustrated in Figure 3(b), the net result of the termination operation is that there are now two black hole processes, swallowing any messages arriving from *A*1 and *A*2. These processes will cease to exist once processes *A*1 and *A*2 each

decided to terminate and send poison down their channels. The result of applying the graceful termination technique to this network with process *B* determining when to terminate, is an incomplete process network termination. Lets investigate how a complete network termination can be achieved using graceful termination and the possibility of every process of the network being able to initiate a complete termination.



(a) Process network ready for complete termination initiated by B

(b) Process network modified to allow termination initiation by any process

Figure 4. Complete terminatable complex process network for different poison originators

With the poison represented as a message, every incoming message needs to be checked whether it is poison or not. This constant checking not only results in an overhead, but also makes the resulting code more complex, due to the additional code necessary for checking and handling. Also, messages can only travel in the direction of the channel. For processes which act as data source, this can result in extra channels, just to transport the termination message. This camouflages the data flow, and makes use of these processes more complex. To be able to terminate the complete process network of figure 3(a), assuming that process *B* injects the poison, it would be necessary to introduce two additional channels. The resulting process network is shown in figure 4(a). When the decision to inject poison can only be made by process *C*, then a total of three additional channels are required. The data flow of the process network becomes totally hidden if we have the goal that any process should be able to initiate a complete network termination, see figure 4(b). Of course the complexity of every process increases as it has to check all channel inputs for incoming messages.

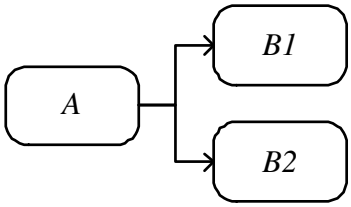


Figure 5. One2Any-Channel with multiple receiver processes

In JCSP, channels are allowed with multiple channel-inputs and outputs, such as the One2Any-, Any2One- or Any2AnyChannels. These channel types are necessary to have a way to implement master-worker environments, where multiple identical workers operate upon requests issued by one master. Another area is that of environments where multiple processes want to submit similar requests to one process. This is, for instance, the case when a process is used to guard a resource from concurrent access. An example for a process network not terminating correctly using this approach, is given in Figure 5. To terminate the process network shown it is necessary to relay a termination message to all processes that are listening on this channel. But how does process *A* know how many processes are listening on the channel?

1.4. Summary of Problems of Graceful Termination and JCSP

The previous sections detailed the problems poisoning poses in general and especially for use in the JCSP environment. In the following our JCSP-Poison approach will be introduced, which tackles these problems:

- Every message received by a process has to be checked, as to whether it is poisonous. For channels that are heavily used, this may waste a lot of computing resources.
- Termination of a process only having an outgoing channel, from outside this process, is not possible, unless a dedicated termination channel is provided. This is caused by delivering the poison as a message, which can only travel in one direction over the channel.
- Partial process network termination is handled by the graceful resetting technique. But requires to adjust all processes connecting to the sub-network.
- JCSP channels can have multiple writer and reader processes, but only one writer and one reader can exchange messages at any one time. To deliver poison in such a system, the sending process would need to know how many processes are connected to each channel over which it is sending poison over. For a process this is impossible to know, as the decision is made by the developer of the overlying process network.

The following section will detail how these problems have been resolved for JCSP. Section 3 will cover the actual implementation and how to use it. A CSP model for the JCSP implementation will be developed in section 4. The paper closes with drawing conclusions and an outlook of further work in the area.

2. JCSP-Poison

The first part of this introduction to JCSP-Poison covers improvements for complete process network termination, with the second part detailing the changes to support partial process network termination.

JCSP-Poison tries to provide clean process network termination for JCSP using the graceful termination technique introduced in section 1.2. In order to make this technique feasible in JCSP, the previously mentioned problems need to be overcome.

The core problem of graceful termination in conjunction with JCSP is that poison is defined as a message. A message can only be sent to one recipient: other processes listening on this channel will be unaware of the situation. The poison also only affects the processes that receive it but not the channels that carry it.

In JCSP-Poison, poison is not a message but an object that is passed from processes to channels and vice versa. A special mechanism for propagation is used. The propagation of poison is done by injecting it into a channel instead of using the normal channel-output methods. A channel which got injected with poison is considered to be poisoned. The delivery of the poison to a process is done by throwing an exception, whenever a process tries to interact with a poisoned channel. This is a similarity to the poisoned channel of Gerald Hilderink, mentioned in [9], where an *ArrrghException* is thrown at anyone, tries to use a poisoned channel. What is unclear here is who is allowed to poison the channel, only the writer end or is it possible for the reader end to do this also. The channels of C++CSP [10] both channel ends are able receive and deliver poison. The direction independence of the injection method, makes it possible for the poison to travel in the reverse direction of the data flow of a channel, thus avoiding the need to add extra channels for this purpose. Once a channel gets poisoned it wakes up any processes currently waiting for a rendezvous and throws a poison exception at them.

This technique solves the following of the earlier mentioned problems:

1. No need to check every incoming message for poison, therefore clearer handling code and less computing resource usage;
2. The poison can propagate backwards over the channels, making correct termination of a process network easier to implement;
3. Due to the channel rejecting any interaction with it, no black hole processes are required to avoid deadlocks. This once again saves computing resources.
4. Poison spreads to all processes using a channel, even when channels with multiple inputs and outputs are used;
5. Handling of poison messages is enforced, due to the delivery as PoisonException and the Java compiler enforcing exception handling.

This technique is optimised to terminate a complete process network. Its working is similar to a bulldozer destroying everything in its path: it terminates every process that crosses its path. In this state it is unsuitable for terminating only parts of a process network. To terminate sub-networks is a requirement for exchanging software modules in an SDR-Platform. An SDR-Platform is basically split into a signal processing part and a data acquisition part. The signal processing part is represented by exchangeable software modules. The data acquisition part runs constantly, in the case of it terminating it terminates the software module as well.

2.1. Poisoning of Sub-Networks

To poison sub-networks it is necessary to limit the propagation of poison in the process network. To do so one could install channels that do not let poison pass, at the borders of the sub network. This approach has the flaw that it effectively prevents the complete termination of the process network. To avoid this, it is necessary for the channels at the borders to be able to differentiate between a complete and a sub-network termination message. This can be achieved by having multiple types of poison, in the case of JCSP-Poison two types are defined:

- GlobalPoison: GlobalPoison is distributed throughout the complete Process Network, it is never filtered out. This type of poison is used to terminate the complete process network.
- LocalPoison: LocalPoison is used to terminate sub-networks and is filtered out by the channels at the borders.

Both types of poison are derived from a common base class. With these two types of poison comes the need to ensure that the type of poison does not change during propagation. Therefore, the PoisonException carries a reference to the original poison exception, which can be retrieved in the exception handler and injected into the other channel ends of the process. The two types of poison and the different channels provided are the main difference between JCSP-Poison and its predecessors.

2.2. Poisonable-Channel

A Poisonable-channel can be in two states. The first one being normal operation, in this state the channel acts as a normal channel. However, once the channel is in the poisoned state, all requests to it, whether reading or writing to it, will result in the channel issuing an exception. To change from normal state into poisoned state, the channel offers the void injectPoison(Poison) method. The name *injectPoison* was chosen because the process that poisons the channel has the choice between two types of poison. Another reason was that it seemed a nice connection to the real world, where poison is injected into an organism.

2.3. Poison-Filtering-Channel

A Poison-Filtering-Channel is a channel that acts as a normal channel, but depending on the type of poison received acts differently. When receiving a GlobalPoison the channel will act like a poisoned Poisonable-Channel, thus allowing for a complete process network termination. The situation is different if a LocalPoison is received. In this case, the poison will not be relayed at all but silently disposed of and the Poison-Filtering-Channel will not change in to the poisoned state. It is the responsibility of the developer not to operate on a channel end previously poisoned. Use of these channels is at the borders of sub-networks, to avoid LocalPoison leaving the sub-network. These channels have to be used with care, as they can easily result in systems that deadlock. This is, for instance, the case when two sub-networks are connected over a Poison-Filtering-Channel and one of these networks terminates using a LocalPoison. The other sub-network not knowing about this termination, might now try to communicate with the terminated network over the channel, but will wait for a reply indefinitely. Poison-Filtering-Channels should only be used in cases where the sub-network is meant to be exchanged. In this case they ensure that no message is lost during the change of the sub-networks.

One feature concerning Poison-Filtering-Channels is when they have multiple channel-inputs or outputs. As they effectively block any LocalPoison arriving and do not relay it at all, it is possible to terminate only one connected sub-network. This allows the number of worker process sub-networks in a master-worker system to be adjusted dynamically.

2.3.1. Poison-Injector-Channel

A special case of Poison-Filtering-Channel is the so called Poison-Injector-Channel. This channel behaves normally like a normal One2One Poison-Filtering-Channel, i.e. no LocalPoison will affect it. But it is possible for a process to poison one channel end, with LocalPoison, using either the injectLocalPoisonIntoWriter() or injectLocalPoisonIntoReader().

This channel makes it possible to inject a LocalPoison into a sub-network to terminate it. At the same time a complete process network termination is not hindered, i.e. no special precautions have to be made.

An implementation of this channel exists, but so far no CSP model has been created.

3. JCSP Implementation

In the previous sections the ideas behind JCSP-Poison were discussed. The following section discusses the implementation of poisoning in JCSP-Poison. It starts with the definition of poison in JCSP, followed by a definition of the interface exposed by the poisonable channels.

3.1. Representation of Poison

In JCSP-Poison, poison is represented by two classes: GlobalPoison and LocalPoison. Both classes are derived from the interface: Poison. This allows the use of functions which are indifferent to the type of poison they handle. The classes representing poison have no further functionality. The class diagram in figure 6 shows the relationship of Poison and its children.

3.2. Delivery of Poison to a Process

In JCSP-Poison, poison is delivered to a process not as a message, but as an exception. The class PoisonException is used for this task. The exception has a member which can hold an instance of Poison. The method Poison getPoison() is used to retrieve the value of this member. This new property requires a redefinition of the JCSP channel ends, detailed in section 3.3.4.

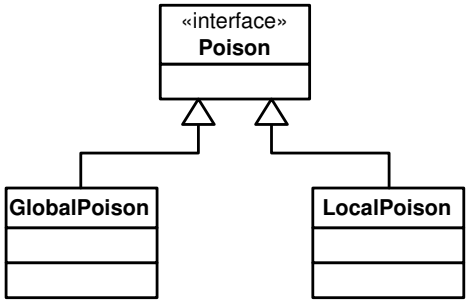


Figure 6. Class Diagram of the different types of Poison

```
public interface Poisonable {
  /**
  * Injects Poison into a Channel;
  * @param Poison Poison to be injected.
  */
  public void injectPoison(Poison poison);
}
```

Listing 1: Definition of the Poisonable interface

3.3. Poisonable-Channel Interface

A JCSP-Poison channel provides three methods to the processes using it:

1. Poisoning the channel.
2. Sending a message.
3. Receiving a message.

In the following these three methods will be detailed.

3.3.1. Poisoning a Channel

For JCSP-Poison to work, it is important to be able to poison a channel. The question is how to do that. With poison not being a message in JCSP-Poison, and its need to travel backwards over a channel, it was not possible to use the normal Object read() and void write(Object) methods. For this reason a new method: void injectPoison(Poison), has been added to the JCSP channel interface. This method is available to both channel ends, accepting any type of poison. Once this method is called, and decided that the channel should be poisoned, it will wake up any process currently waiting for a channel transaction, and a poison exception will be thrown.

The implementation of the void injectPoison(Poison) differs between the Poisonable- and Poison-Filtering-Channels. Poison-Filtering-Channels only become poisoned by a GlobalPoison. This is the only difference between these two JCSP channel types.

Poisonable Interface: As both channel ends allow for injecting poison into the channel an interface called Poisonable has been defined which contains the definition of the injectPoison(Poison poison) method. Its source is given in listing 1.

3.3.2. Sending a Message

In JCSP sending a message is done using the void write(Object object) method. To be able to deliver the poison exception, this method had to be enabled to throw the PoisonException.

```
public interface PoisonableChannelOutput extends Poisonable {
    /**
     * This method sends a message over a channel.
     * @param object
     *        Reference to the message to send.
     * @throws PoisonException
     *         Is thrown when the channel has been poisoned
     */
    public void write(Object object) throws PoisonException;
}
```

Listing 2: The PoisonableChannelOutput interface

```
public interface PoisonableChannelInput extends Poisonable {
    /**
     * This method reads a message from the channel.
     * @return The message read from the channel
     * @throws PoisonException
     *         Is thrown, when the channel has been poisoned.
     */
    public Object read() throws PoisonException;
}
```

Listing 3: The PoisonableChannelInput interface

3.3.3. Receiving a message

To receive messages from a channel in JCSP the method Object read() is used. Like the write method previously this method had to be extended to be able to throw exceptions of type PoisonException.

3.3.4. Channel Ends

A channel in CSP has two ends, a writer and a reader channel end. In the following paragraphs the definition of these channel ends in JCSP-Poison is detailed.

Writer Channel End: The complete writer channel end for the Poisonable-Channel is given in listing 2. To allow the writer process to poison the channel, this interface is derived from the Poisonable interface of listing 1.

Reader Channel End: The reader channel end of a JCSP-Poison channel is a combination of the Poisonable interface and the modified read method. Its definition is shown in listing 3.

Relationship of Channels and Channel Ends: The class tree of figure 7 shows the relationship between the three interfaces defined previously. This concludes the definition of the interface exposed by the JCSP-Poison channels. A derivation of the poisonable channels from the standard JCSP channels removes the distinction between the normal channels and poisonable channel ends. It is therefore not advisable. It might seem to be a good idea from the perspective of code reuse. This is actually not the case, as all methods of the original JCSP channel implementation had to be modified, so no code could be reused by derivation. The JCSP channel implementation code was reused by means of copy and paste, so the system did not start from scratch. Only the implementation of the Poisonable-Channel and Poison-Filtering-Channel constructs were added to the system, leaving the rest of JCSP untouched.

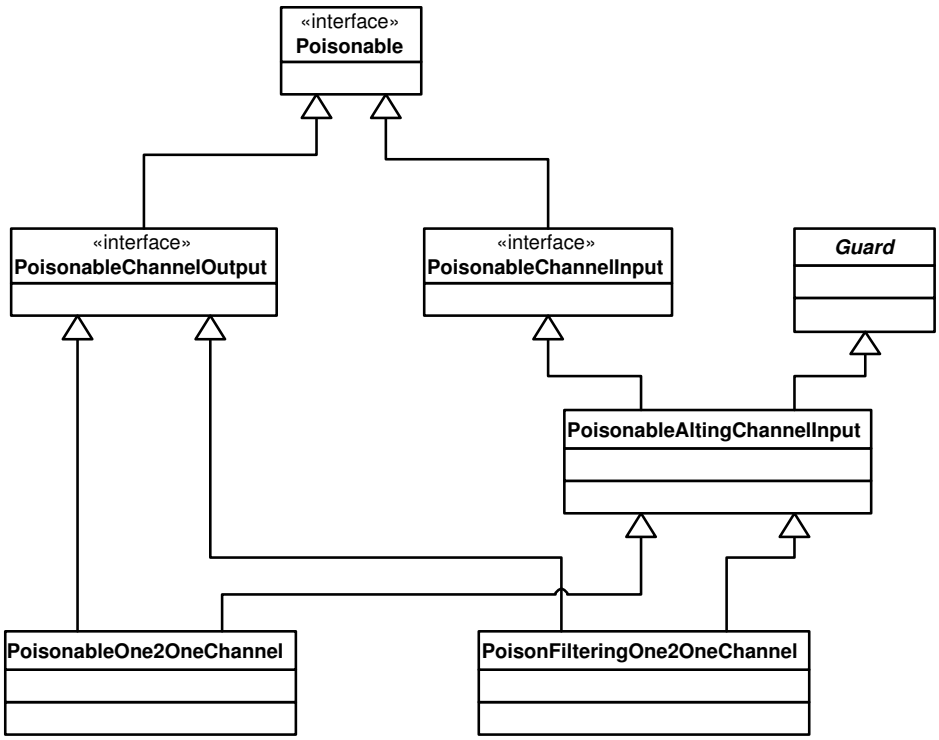


Figure 7. Class diagram of the Poisonable- and the Poison-Filtering-Channel

3.3.5. Provided Channels

JCSP-Poison provides currently three channel implementations:

- The class PoisonableOne2OneChannel provides an implementation of the Poisonable-Channel defined earlier.
- The Poison-Filtering-Channel is provided by the class PoisonFilteringOne2OneChannel.
- The Poison-Injector-Channel is provided by the class PoisonInjectorOne2OneChannel

The class diagram of the two channel types is given in figure 7, these implementations have been modelled using CSPM by us. These models are not shown here, due to their high complexity. Both channel models have been tested for equivalence with the models shown in section 4, using FDR 2.80 [11]. There are also unchecked versions of the One2Any, Any2One- and Any2AnyChannels available, which should work fine, since they are derived from the One2OneChannel versions. At least during the test phase we could not find any anomalies, but admit that this is no proof for correctness of implementation.

3.4. An Example for Handling of Poison

To give a simple example of how JCSP-Poison is used in practice, one of the test processes is discussed. This process is not doing anything useful but shows how a process should react upon catching a PoisonException.

The process of listing 4 has two PoisonableChannelOutputs: `out1` and `out2`. While running, this process sends integers of increasing value over the two channels connected to it. If one of these channels gets poisoned, it reports this to the process, by throwing an exception of type PoisonException, once the process tries to access the channel. The process catches this exception and injects the received poison into both channels, using the injectPoison(Poison) method. The poison that has been transferred over the channel is retrieved from the exception

```
public class Producer implements CSProcess {
    PoisonableChannelOutput out1;
    PoisonableChannelOutput out2;

    public Producer(PoisonableChannelOutput out1,
                    PoisonableChannelOutput out2) {
        this.out1 = out1;
        this.out2 = out2;
    }
    public void run() {
        Integer inputInt = null;
        int i = 0;
        while (true) {
            try {
                i++;
                out1.write(new Integer(i));
                out2.write(new Integer(i));
            } catch (PoisonException e) {
                out1.injectPoison(e.getPoison());
                out2.injectPoison(e.getPoison());
                return;
            }
        }
    }
}
```

Listing 4: Example of a JCSP Process handling a PoisonException

using the method Poison getPoison(). It is important to mention that no process should create its own instance of a poison while handling a PoisonException.

The example given shows that JCSP-Poison is easy to apply in JCSP based programs. Also altering of existing JCSP process networks should be easy without disrupting existing structures.

4. CSP Model for the Poisonable Channels

In the following, the CSP model for the poisonable channels is developed. It models the functionality of the poisonable channels Java implementation. It is not a direct mapping of the Java code, but only behaves like its Java counterpart. The counter part is a direct mapping of the Java implementation of the PoisonableOne2OneChannel and PoisonFilteringOne2OneChannel classes to CSP. This has been developed, but is not shown in this paper. The reason for this is the high complexity of this model due to trying to be as close to Java as possible. The Java implementation model is equivalent to the model shown here.

The section starts with defining the interface used to communicate with the channel. This is followed by the development of the CSP model of the Poisonable-Channel and the Poison-Filtering-Channel. We know that the model shown has a high complexity, it is caused by the fact that this model has been developed to match its java counter part. Plans to develop a simplified model exist, but have not yet been implemented.

4.1. Interface of the Poisonable Channels

Channels in JCSP, due to their unidirectional nature, have a writer end and a reader end. This results in two different protocols, one for each end. As discussed earlier, the goal of

the poisonable channels is to allow poison to travel backwards over channels, to get an easy and therefore safe to use poison mechanism. Furthermore, the poison should not be encoded in standard messages, but should be reported by using the Java exception mechanism. The protocols used for these channels are orientated by the way Java method calls are modelled in [12]. The reason for this is that the model shown here is the counter part to the CSP version of the Java implementation for the poisonable channels. The model given in [12] has been extended by us to support exceptions and also to standardise the names of the channels and events used to perform function calls. It is not included in this paper, but whenever this paper drifts away from the original model a short explanation is given.

The protocols shown here are meant for interaction with the model given for the poisonable channels. That is the reason why these processes terminate after having sent a message.

Before defining the protocol for reader and writer in detail, the data types for the message and the poison have to be defined, this is done in equation 1. This equation defines three sets: *Poison*, *InternalPoison* and *Data*. The set *Poison* defines the types of poison used outside the channel and consists of *LocalPoison* and *GlobalPoison*. The channel itself can be in three possible states: not poisoned *None*, poisoned with *LocalPoison* and poisoned with *GlobalPoison*. These states are stored in a variable of type *InternalPoison*, which incorporates *Poison* and *None*. The *Data* set defines the possible messages that can be transferred over the channels.

$$\begin{aligned} \mathit{Poison} &= \{\mathit{LocalPoison}, \mathit{GlobalPoison}\} \\ \mathit{InternalPoison} &= \mathit{Poison} \cup \{\mathit{None}\} \\ \mathit{Data} &= \{\mathit{True}, \mathit{False}\} \end{aligned} \tag{1}$$

Due to the fact that the model is meant to provide a functional model of a Java implementation, it is necessary to include the notion of object and thread into the model. This is done by appending the object-id and thread-id to the channel and event names. The sets of equation 2 define the sets *Objects* for possible object-ids and *Threads* for the thread-ids. In the present model there is always only object of a channel being created. Therefore, *Objects* containing only one object-id is sufficient. If at a later time the number of possible objects should be increased, the set *Objects* has to be modified. To be able to check a channel it is necessary to have at least two processes accessing it, in JCSP processes are represented by threads, therefore at least two thread-ids have to be available.

$$\begin{aligned} \mathit{Objects} &= \{0\} \\ \mathit{Threads} &= \{1, 2\} \end{aligned} \tag{2}$$

4.1.1. Modelling Method Calls and Exceptions

In the model used in this paper, methods are called by an event *methodname_start*.*o*.*t*, with *o* being the object-id and *t* the thread-id used. If the method has one parameter, this is transferred over a channel named *methodname_start*.*o*.*t*: transmission of the parameter will then also start the method. Once the method has completed its task, it will acknowledge the successful transaction using an event of name *methodname_ack*.*o*.*t*. If the method is provides a return value, this value will be transferred over a channel of the same name.

Exception will be transferred over a channel named *methodname_ex*.*o*.*t*. An exception can be thrown by a method during the whole time the process is waiting for the *methodname_ack*.*o*.*t* event. To be able to cater for that, the reception of the exception message is modelled as an interruption.

4.1.2. Inject Poison Interface

Both reader and writer sides can inject poison into a channel. Injecting poison is a time consuming operation, and no other operations should take place at the same time. To cater for this, it is modelled as a method call using these two events:

- *inject_poison_start*.*o*.*t*.*p* – thread *t* : *Threads* wants to inject poison *p* : *Poison* into the channel with the object-id *o* : *Objects*;
- *inject_poison_ack*.*o*.*t* – injecting poison into the channel with object-id *o* : *Objects* by thread *t* : *Threads* completes successfully;

The resulting alphabet α*INJECT_POISON*$(o : \mathit{Objects}, t : \mathit{Threads})$ is given in equation 3. Processes can not know whether they will receive a *LocalPoison* or a *GlobalPoison*, therefore this alphabet includes both types of poison.

$$\begin{aligned} &\alpha\mathit{INJECT\_POISON}(o : \mathit{Objects}, t : \mathit{Threads}) = \\ &\quad \{\mathit{inject\_poison\_start}.o.t.p, \mathit{inject\_poison\_ack}.o.t \mid p \in \mathit{Poison}\} \end{aligned} \tag{3}$$

The snippet of equation 4, shows how to inject a poison *p* : *Poison* into a Poisonable-Channel *o* : *Objects* by a thread *t* : *Threads*. Developers wanting to use this functionality can simply adjust it to their needs and insert in their own code.

$$\mathit{inject\_poison\_start}.o.t!p \rightarrow \mathit{inject\_poison\_ack}.o.t \rightarrow \ldots \tag{4}$$

4.1.3. Write Interface

The write method takes the message *m* : *Data* to be transferred as parameter *write_start*.*o*.*t*!*m*, and successful completion of the transaction is signalled by the event *write_ack*.*o*.*t*. If the channel is poisoned, the poison will be transferred over the channel *write_ex*.*o*.*t*. In equation 6 the CSP model of a write method call is given.

- *write_start*.*o*.*t*.*m* – thread *t* : *Threads* wants to write a message *m* : *Data* on a channel *o* : *Objects*;
- *write_ack*.*o*.*t* – the write operation, by thread *t* : *Threads*, on the channel *o* : *Objects*, completed successfully;
- *write_ex*.*o*.*t*.*p* – the write operation of thread *t* : *Threads*, on channel *o* : *Objects* was interrupted by a Poison-Exception, delivering poison *p* : *Poison*;

In equation 5, the alphabet α*WRITE*$(o : \mathit{Objects}, t : \mathit{Threads})$ is given. It should be incorporated by processes that want to write to poisonable channels.

$$\begin{aligned} &\alpha\mathit{WRITE}(o : \mathit{Objects}, t : \mathit{Threads}) = \\ &\quad \{\mathit{write\_start}.o.t.m, \mathit{write\_ack}.o.t, \mathit{write\_ex}.o.t.p \mid m \in \mathit{Data}, p \in \mathit{Poison}\} \end{aligned} \tag{5}$$

The code snippet of equation 6, can be used when writing to poisonable channels. It is important that the user of the poisonable channels offers the ability to receive exceptions delivered by the *write_ex*.*o*.*t* channel.

$$\begin{aligned} &(\mathit{write\_start}.o.t!m \rightarrow \mathit{write\_ack}.o.t \rightarrow \ldots) \\ &\triangle\ (\mathit{write\_ex}.o.t?p : \mathit{Poison} \rightarrow \ldots) \end{aligned} \tag{6}$$

4.1.4. Read Interface

The protocol to be used on the reader side is given in equation 8. Similar to the writer protocol the reader protocol can be interrupted by an exception if the channel is poisonous.

- *read_start.o.t* – thread t : *Threads* wants to read a message from a Poisonable-Channel o : *Objects*;
- *read_ack.o.t.m* – thread t : *Threads* read operation, on the channel o : *Objects* terminates successfully, returning message m : *Data*;
- *read_ex.o.t.p* – the read operation of thread t : *Threads*, on channel o : *Objects* resulted in a Poison-Exception to be thrown with poison p : *Poison*;

The alphabet of the *READ* process, given in equation 5, contains all events that are possible in when reading from a Poisonable-Channel o : *Objects* with a thread t : *Threads*.

$$\begin{aligned} &\alpha READ(o : Objects, t : Threads) = \\ &\quad \{read\_start.o.t, read\_ack.o.t.m, read\_ex.o.t.p \mid m \in Data, p \in Poison\} \end{aligned} \tag{7}$$

Reading from a Poisonable-Channel requires similar precautions as writing to it. The code snippet given in equation 8, shows how to do it correctly.

$$\begin{aligned} &(read\_start.o.t \rightarrow read\_ack.o.t!m : Data \rightarrow \dots) \\ &\triangle\ (read\_ex.o.t?p : Poison \rightarrow \dots) \end{aligned} \tag{8}$$

4.1.5. The Reader and Writer Channel Ends

Traditionally the writer end of a channel is considered to be the left side, while the reader end is the right side. In our model a Poisonable-Channel offers its writer and reader ends to inject poison into the channel. This has to be reflected in the interface of the poisonable channels. The interface for the left side of a Poisonable-Channel is given in equation 9, while the interface of the right side is given in equation 10.

$$\begin{aligned} &\alpha LEFT(o : Objects, t : Threads) = \\ &\quad \alpha WRITE(o,t) \cup \alpha INJECT\_POISON(o,t) \end{aligned} \tag{9}$$

$$\begin{aligned} &\alpha RIGHT(o : Objects, t : Threads) = \\ &\quad \alpha READ(o,t) \cup \alpha INJECT\_POISON(o,t) \end{aligned} \tag{10}$$

4.1.6. The Complete Poisonable Channels Interface

The complete interface, valid for all poisonable channels: $\alpha PC(o,t_1,t_2)$, is obtained by combining the $\alpha LEFT(o,t)$ and $\alpha RIGHT(o,t)$. The resulting interface αPC, is given in equation 11 and expanded in equation 12. Figure 8 gives a graphical representation of the interface poisonable channels represented by the process $PC(...)$ expose.

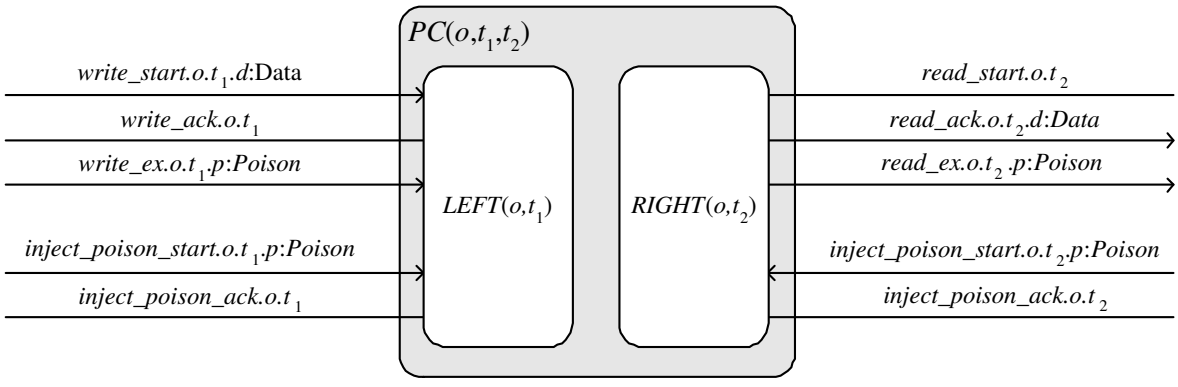


Figure 8. Interface for all Poisonable Channels

$$\begin{aligned} &\alpha PC(o : Objects, t_1 : Threads, t_2 : Threads) = \\ &\quad \alpha LEFT(o, t_1) \cup \alpha RIGHT(o, t_2) \end{aligned} \tag{11}$$

$$\begin{aligned} &\alpha PC(o, t_1, t_2) = \\ &\left\{ \begin{array}{l} write\_start.o.t_1.d, write\_ack.o.t_1, inject\_poison\_ack.o.t_1, \\ inject\_poison\_start.o.t_1.p, read\_ex.o.t_1.p, \\ read\_start.o.t_2, read\_ack.o.t_2.d, inject\_poison\_ack.o.t_2, \\ inject\_poison\_start.o.t_2.p, write\_ex.o.t_2.p \\ \mid d \in Data, p \in Poison \end{array} \right\} \end{aligned} \tag{12}$$

4.2. Test-Bench for Poisonable Channels

In the following the desired behaviour of the poisonable channels is defined. This is necessary to test whether the model developed later on complies with our expectations. For this purpose the test-bench given in equations 13 till 19has been created. The test-bench covers the different scenarios the Poisonable-Channel and the Poison-Filtering-Channel should cater for. In section 4.5 the channel models are verified using this test-bench.

4.2.1. Sending a Message

The Poisonable channels should operate just like normal channels. Equation 13 defines the process *TB_MESSAGE_TRANSFER*$(...)$, which specifies a normal channel transaction using the interface specified above.

$$\begin{aligned} &TB\_MESSAGE\_TRANSFER(o : Objects, t_1 : Threads, t_2 : Threads, m : Data) = \\ &\quad TB\_WRITE\_MESSAGE(o, t_1, m) \\ &\quad |[\, \alpha TB\_WRITE\_MESSAGE(o, t_1, m) \,|\, \alpha TB\_READ\_MESSAGE(o, t_2) \,]| \\ &\quad TB\_READ\_MESSAGE(o, t_2) \setminus \{transmit.o.d \mid d \in Data\} \end{aligned} \tag{13}$$

$$\begin{aligned} &TB\_WRITE\_MESSAGE(o : Objects, t : Threads, m : Data) = \\ &\quad write\_start.o.t.m \rightarrow transmit.o!m \rightarrow write\_ack.o.t \rightarrow TB\_WRITE\_MESSAGE(o, t, m) \\ &\alpha TB\_WRITE\_MESSAGE(o : Objects, t : Threads, m : Data) = \\ &\quad \{write\_start.o.t.m, write\_ack.o.t, transmit.o.d \mid d \in Data\} \end{aligned} \tag{14}$$

The *TB_WRITE_MESSAGE*$(...)$ process of equation 14 performs a successful write transaction on a channel, complying to the interface specification of the poisonable channels.

$$\begin{aligned} &TB\_READ\_MESSAGE(o : Objects, t : Threads) = \\ &\quad read\_start.o.t \rightarrow transmit.o?x \rightarrow read\_ack.o.t.x \rightarrow TB\_READ\_MESSAGE(o, t, m) \\ &\alpha TB\_READ\_MESSAGE(o : Objects, t : Threads, m : Data) = \\ &\quad \{read\_start.o.t, read\_ack.o.t.d, transmit.o.d \mid d \in Data\} \end{aligned} \tag{15}$$

In equation 15 the successful reception of messages is modelled, according to the poisonable channels interface specification.

4.2.2. Sending a Message, Reader Injects Poison

The process of equation 16 simulates the case, that the writer end of the channel tries to write a message, while reader end injects poison into the channel. In this case the writer should receive an exception.

$$\begin{aligned} &TB\_READER\_INJECTS\_POISON(o : Objects, t_1 : Threads, t_2 : Threads, p : Poison) = \\ &\quad write\_start.o.t_1.True \rightarrow inject\_poison\_start.o.t_2.p \rightarrow \\ &\quad\quad inject\_poison\_ack.o.t_2 \rightarrow write\_ex.o.t_1.p \rightarrow STOP \end{aligned} \tag{16}$$

4.2.3. Trying to Receive a Message, Writer Injects Poison

In equation 17 the roles are reversed, now the reader end tries to receive a message, while the writer end injects poison into the channel. The reader should receive an exception.

$$\begin{aligned} &TB\_WRITER\_INJECTS\_POISON(o : Objects, t_1 : Threads, t_2 : Threads, p : Poison) = \\ &\quad read\_start.o.t_2 \rightarrow inject\_poison\_start.o.t_1.p \rightarrow \\ &\quad\quad inject\_poison\_ack.o.t_1 \rightarrow read\_ex.o.t_2.p \rightarrow STOP \end{aligned} \tag{17}$$

4.2.4. Access to a Poisoned Channel

Process $TB\_POISONED\_CHANNEL(\ldots)$, given in equation 18, first poisons the channel, and then lets both reader and writer ends access it. It is expected that both processes receive an exception.

$$\begin{aligned} &TB\_POISONED\_CHANNEL(o : Objects, t_1 : Threads, t_2 : Threads, p : Poison) = \\ &\quad inject\_poison\_start.o.t_1.p \rightarrow inject\_poison\_ack.o.t_1 \rightarrow \\ &\quad\quad read\_start.o.t_2 \rightarrow read\_ex.o.t_2.p \rightarrow \\ &\quad\quad write\_start.o.t_1.True \rightarrow write\_ex.o.t_1.p \rightarrow STOP \end{aligned} \tag{18}$$

4.2.5. A LocalPoisoned Channel is Used

$TB\_LOCAL\_POISONED\_CHANNEL(\ldots)$ of equation 19 demonstrates that a Poison-Filtering-Channel really filters out a *LocalPoison* and operates normally after that. This is done by first injecting *LocalPoison* into the channel and then performing normal channel operations using the $TB\_MESSAGE\_TRANSFER(\ldots)$ process.

$$\begin{aligned} &TB\_LOCAL\_POISONED\_CHANNEL(o : Objects, t_1 : Threads, t_2 : Threads, m : Data) = \\ &\quad inject\_poison\_start.o.t_1.LocalPoison \rightarrow inject\_poison\_ack.o.t_1 \rightarrow \\ &\quad\quad TB\_MESSAGE\_TRANSFER(o, t_1, t_2, m) \end{aligned} \tag{19}$$

4.3. The Poisonable-Channel

The Poisonable-Channel's behaviour depends upon the state it is in. It can be in any of these three states:

1. Normal: In this state it operates as a normal CSP channel. This is the default state after creation of the channel. When injected with *LocalPoison* the channel will change into the LPoison state. Injecting the channel with *GlobalPoison* will change the state to GPoison.
2. LPoison: The channel changes into this state after it gets injected with *LocalPoison* in the Normal state. Every invocation of either read or write functions will be answered by an exception with *LocalPoison*. The channel stays in this state until it gets injected with *GlobalPoison*, then it changes into the GPoison state.
3. GPoison: In this state the channel will answer any read or write request by throwing an exception with *GlobalPoison*. There is no possible state change from this state.

4.3.1. The POISON(o : Objects) Process

The $POISON(o : Objects)$ process is responsible to store the state of the poisonable channels. With poison being injectable from both sides of a channel it is necessary to provide both sides with access to this process. The set *PoisonAccess* of equation 20, defines the possible accessors of the process.

$$PoisonAccess = \{left, right\} \tag{20}$$

The state of the channel has to be consistent for both sides of the channel. This means that once one side of the channel receives a poison, both channel ends deliver it to their users. To keep the channel state consistent only one side of the channel is allowed to change the poison state of the channel at a time, following the CREW rules. The read access has therefore, be blocked during a change of the channel state. In the $POISON(o : Objects)$ process this is achieved by going into a locked state, using the $poison\_lock.o.a : PoisonAccess$ events, before altering the state of the channel. Once the state change is done the process is unlocked with the $poison\_unlock.o.a : PoisonAccess$ events.

Equations 21 - 25, give the definition of the $POISON(o : Objects)$ process. Figure 9 illustrates the interface the $POISON(o : Objects)$ process offers to its environment.

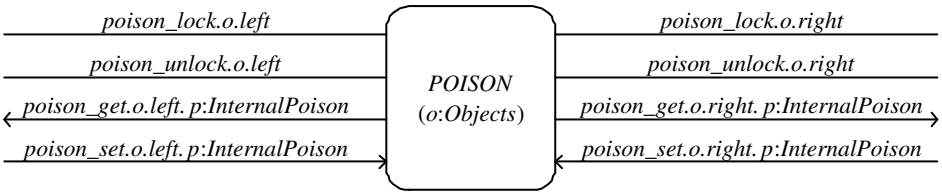


Figure 9. Interface of the POISON(o:Objects) process

$$\begin{aligned} &POISON(o : Objects) = \\ &\left(\Box_{a:PoisonAccess} poison\_lock.o.a \rightarrow POISON\_LOCKED(o, a, POISON(o))\right) \\ &\Box \left(\Box_{a:PoisonAccess} poison\_get.o.a!None \rightarrow POISON(o)\right) \end{aligned} \tag{21}$$

$$LPOISON(o : Objects) =$$

$$\left(\Box_{a:PoisonAccess} poison\_lock.a \rightarrow POISON\_LOCKED(a, LPOISON(o))\right) \quad (22)$$
$$\Box \left(\Box_{a:PoisonAccess} poison\_get.o.a!LocalPoison \rightarrow LPOISON(o)\right)$$

$$GPOISON(o : Objects) =$$

$$\left(\Box_{a:PoisonAccess} poison\_lock.o.a \rightarrow POISON\_LOCKED(a, GPOISON(o))\right) \quad (23)$$
$$\Box \left(\Box_{a:PoisonAccess} poison\_get.o.a!GlobalPoison \rightarrow GPOISON(o)\right)$$

$$POISON\_LOCKED(o : Objects, a : PoisonAccess, callingProcess) =$$
$$\left(poison\_set.a.None \rightarrow poison\_unlock.o.a \rightarrow callingProcess\right)$$
$$\Box \left(poison\_set.o.a.LocalPoison \rightarrow poison\_unlock.o.a \rightarrow LPOISON(o)\right) \quad (24)$$
$$\Box \left(poison\_set.o.a.GlobalPoison \rightarrow poison\_unlock.o.a \rightarrow GPOISON(o)\right)$$
$$\Box \left(poison\_unlock.o.a \rightarrow callingProcess\right)$$

$$\alpha POISON(o : Objects) = \left\{ \begin{array}{l} poison\_get.o.a.p, poison\_set.o.a.p, \\ poison\_lock.o.a, poison\_unlock.o.a \\ \mid a \in PoisonAccess, p \in InternalPoison \end{array} \right\} \quad (25)$$

4.3.2. The POISON_VALVE(...) Process

In the previous section the states of the poisonable channels and their handling were detailed. So far it is possible to correctly handle poison that is received while both channel ends are unused. But what if one end tries to send or receive a message? In this case the waiting channel end should be woken up and throw an exception with the received poison at its user. This requires that the channel end injected with poison, sends a wake up call to the waiting channel end. But this is only necessary when one channel end is currently waiting. Therefore it is necessary for the channel ends to be able to filter events when they are not appropriate. This task is done by the *POISON_VALVE*(...) process, which is detailed in equations 27 till 30, a graphical representation is given in figure 10. To control the valve, a channel carrying messages of type *ValveControl* is used, the type is defined in equation 26.

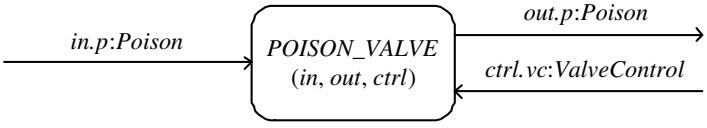


Figure 10. Interface of the *POISON_VALVE*(...) process

$$ValveControl = \{open, close\} \quad (26)$$

$$POISON\_VALVE(in, out, ctrl) = POISON\_VALVE\_CLOSED(in, out, ctrl) \quad (27)$$

$$
\begin{aligned}
&POISON\_VALVE\_OPEN(in, out, ctrl) = \\
&\quad \big(ctrl.close \rightarrow POISON\_VALVE\_CLOSED(in, out, ctrl)\big) \\
&\quad \Box \left(\begin{array}{l} in?m \rightarrow \\ \quad \left(\begin{array}{l} \big(out!m \rightarrow POISON\_VALVE\_OPEN(in, out, ctrl)\big) \\ \quad \Box \ \big(ctrl.close \rightarrow POISON\_VALVE\_CLOSED(in, out, ctrl)\big) \end{array} \right) \end{array} \right) \\
&\quad \Box \ \big(ctrl.open \rightarrow POISON\_VALVE\_OPEN(in, out, ctrl)\big)
\end{aligned}
\tag{28}
$$

$$
\begin{aligned}
&POISON\_VALVE\_CLOSED(in, out, ctrl) = \\
&\quad \big(in?m \rightarrow POISON\_VALVE\_CLOSED(in, out, ctrl)\big) \\
&\quad \Box \ \big(ctrl.open \rightarrow POISON\_VALVE\_OPEN(in, out, ctrl)\big) \\
&\quad \Box \ \big(ctrl.close \rightarrow POISON\_VALVE\_CLOSED(in, out, ctrl)\big)
\end{aligned}
\tag{29}
$$

$$
\alpha POISON\_VALVE(in, out, ctrl) = \left\{ \begin{array}{l} in.p, out.p, ctrl.vc \\ \quad \mid p \in Poison, vp \in ValveControl \end{array} \right\} \tag{30}
$$

4.3.3. The PCM_INJECT_POISON(...) Process

Both channel ends offer the environment to inject poison into the channel. As they behave similarly in this respect they both use the *PCM_INJECT_POISON*$(...)$ process for this task. The prefix *PCM* indicates that this process is part of the Poisonable-Channel-Model. The key task of this process is to pass the received poison to the *POISON*$(...)$ process and signal the other channel end about the arrival of new poison, by sending it over the channel given in parameter *pc*. The definition of the process is given in equations 31 and 32, while the interface is illustrated in figure 11.

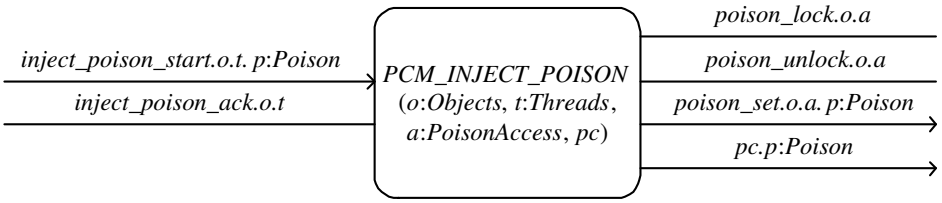


Figure 11. Interface of the *PCM_INJECT_POISON* process

$$
\begin{aligned}
&PCM\_INJECT\_POISON(o : Objects, t : Threads, a : PoisonAccess, pc) = \\
&\quad inject\_poison\_start.o.t?p : Poison \rightarrow poison\_lock.o.a \rightarrow \\
&\quad poison\_set.o.a!p \rightarrow poison\_unlock.o.a \rightarrow pc!p \rightarrow \\
&\quad inject\_poison\_ack.o.t \rightarrow SKIP
\end{aligned}
\tag{31}
$$

$$
\begin{aligned}
&\alpha PCM\_INJECT\_POISON(o : Objects, t : Threads, a : PoisonAccess, pc) = \\
&\left\{ \begin{array}{l} inject\_poison\_start.o.t.p, inject\_poison\_ack.o.t, poison\_lock.o.a, \\ poison\_unlock.o.a, poison\_set.o.a.p, pc.p \mid p \in Poison \end{array} \right\}
\end{aligned}
\tag{32}
$$

4.3.4. The *PCM_TRANSMIT*$(\ldots)$ Process

To allow a channel transaction to take place, a transmitter and a receiver are necessary. The *PCM_TRANSMIT*$(\ldots)$ process, defined in equations 33 and 34, complies to the write interface defined earlier. The complete interface of the process is shown in figure 12.

Before the *PCM_TRANSMIT*$(\ldots)$ process sends the message passed to it over the *write_start*.*o*.*t* channel, it checks whether the channel is poisoned. If that is the case the poison is delivered to the caller over the *write_ex*.*o*.*t* channel. If the reader channel end gets poisoned while the write operation on the channel *transmit*.*o* is still pending, the writer channel end should be alarmed. This is done by having an external choice between the writing on the *transmit*.*o* channel and receiving poison from the *rp*2.*o* channel, rp stands for right side poison. The *rp*2.*o* channel is the output of a *POISON_VALVE*$(\ldots)$ process, discussed earlier. The status of the valve is controlled using the *rp_ctrl*.*o* channel, which stands for right poison control.

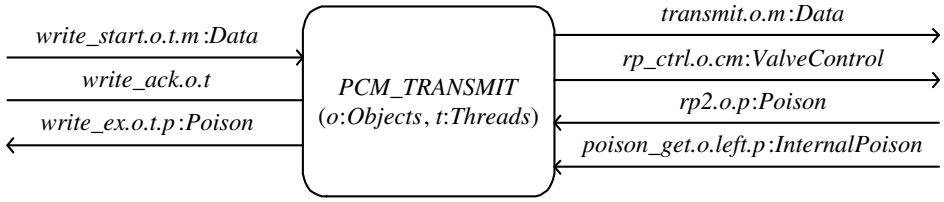


Figure 12. Interface of the *PCM_TRANSMIT* process

$$
\begin{aligned}
&PCM\_TRANSMIT(o : Objects, t : Threads) = \\
&\quad write\_start.o.t?m : Data \rightarrow rp\_ctrl.open \rightarrow poison\_get.left?p \rightarrow \\
&\quad \left(\begin{array}{l}
\textbf{if } p = None \textbf{ then} \\
\quad \left(\begin{array}{l} transmit.o!m \rightarrow rp\_ctrl.o.close \rightarrow \\ write\_ack.o.t \rightarrow SKIP \end{array} \right) \\
\quad \Box \left(\begin{array}{l} rp2.o?p \rightarrow rp\_ctrl.o.close \rightarrow \\ write\_ex.o.t!p \rightarrow SKIP \end{array} \right) \\
\textbf{else} \\
\quad rp\_ctrl.o.close \rightarrow \\
\quad write\_ex.o.t!p \rightarrow \\
\quad SKIP
\end{array} \right)
\end{aligned}
\tag{33}
$$

$$
\begin{aligned}
&\alpha PCM\_TRANSMIT(o : Objects, t : Threads) = \\
&\quad \left\{ \begin{array}{l}
write\_start.o.t.m, write\_ack.o.t, write\_ex.o.t.p, \\
rp\_ctrl.o.open, rp\_ctrl.o.close, rp2.o.p, \\
poison\_get.o.left.ip, transmit.o.m \\
\quad \mid m \in Data, ip \in InternalPoison, p \in Poison
\end{array} \right\}
\end{aligned}
\tag{34}
$$

4.3.5. Preventing Concurrent Writing and Poisoning

The left channel side offers its user two possible operations, either transmission of a message or inject poison into the channel. Both operations can not happen concurrently. To represent this the process $PCM\_LEFT\_CHOOSER(o,t)$ has been created which is an external choice between the $PCM\_TRANSMIT(o,t)$ and the $PCM\_INJECT\_POISON(...)$ processes, its definition is given in equations 35 and 36. A compositional block diagram of the resulting process is given in figure 13.

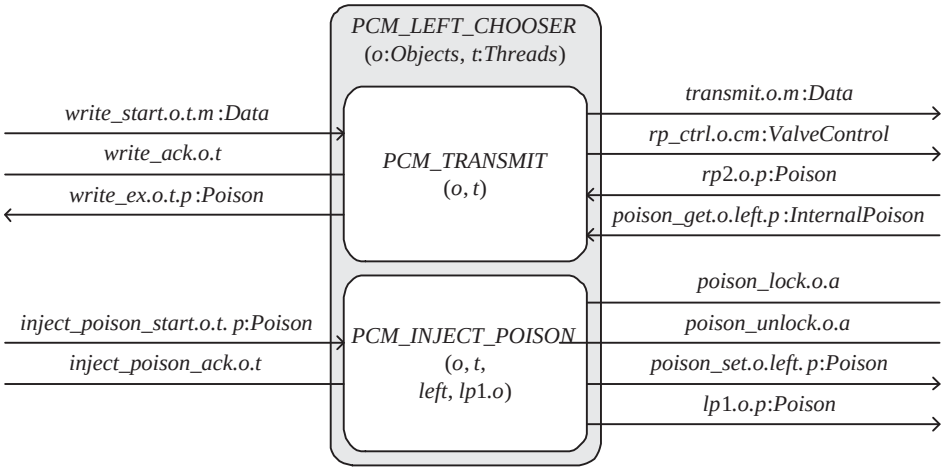


Figure 13. The *PCM_LEFT_CHOOSER* processes composition

$$\begin{aligned} &PCM\_LEFT\_CHOOSER(o,t) = \\ &\quad (PCM\_TRANSMIT(o,t) \;\Box\; PCM\_INJECT\_POISON(o,t,left,lp1.o)) \,; \\ &\quad PCM\_LEFT\_CHOOSER(o,t) \end{aligned} \tag{35}$$

$$\begin{aligned} &\alpha PCM\_LEFT\_CHOOSER(o : Objects, t : Threads) = \\ &\quad \alpha PCM\_TRANSMIT(o,t) \cup \alpha PCM\_INJECT\_POISON(o,t,left,lp1.o) \end{aligned} \tag{36}$$

4.3.6. Writer Channel End

To complete the writer channel end it is necessary to include the $POISON\_VALVE(...)$ process required by the $PCM\_TRANSMIT(...)$ process. The valve can not be brought in earlier, as the valve has to be constantly running, while there is an external choice whether the $PCM\_TRANSMIT(...)$ or the $PCM\_INJECT\_POISON(...)$ process is executed. The $PCM\_LEFT(...)$ process, of equations 37 and 38, is the result of these considerations. The structure of the $PCM\_LEFT(...)$ process is also shown in figure 14.

$$\begin{aligned} &PCM\_LEFT(o : Objects, t : Threads) = \\ &\quad PCM\_LEFT\_CHOOSER(o,t) \\ &\quad |[\, \alpha PCM\_LEFT\_CHOOSER(o,t) \,|\, \alpha POISON\_VALVE(rp1.o, rp2.o, rp\_ctrl.o) \,]| \\ &\quad POISON\_VALVE(rp1.o, rp2.o, rp\_ctrl.o) \end{aligned} \tag{37}$$

$$\begin{aligned} &\alpha PCM\_LEFT(o : Objects, t : Threads) = \\ &\quad \alpha PCM\_LEFT\_CHOOSER(o,t) \cup \alpha POISON\_VALVE(rp1.o, rp2.o, rp\_ctrl.o) \end{aligned} \tag{38}$$

The writer channel end of the Poisonable-Channel is now completely defined.

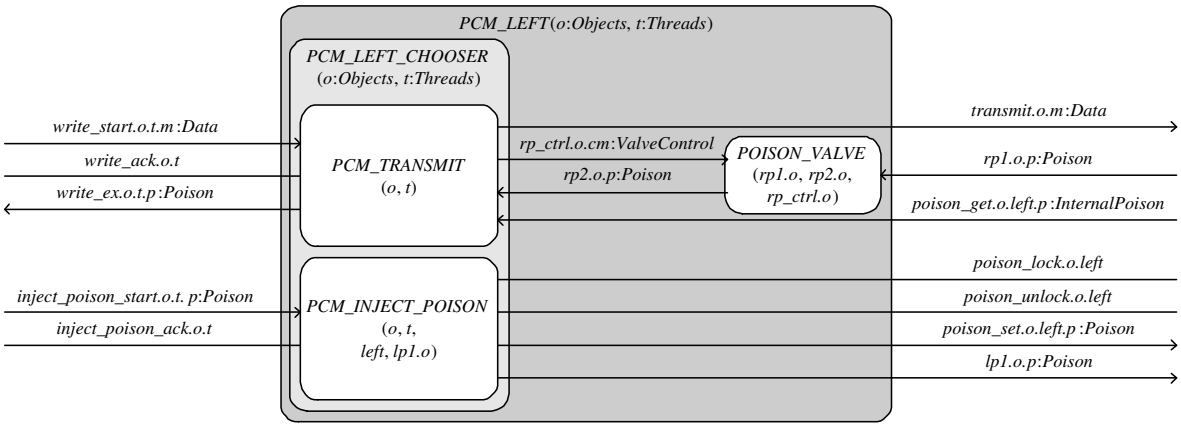


Figure 14. Interface of the *PCM_LEFT* process

4.3.7. PCM_RECEIVE(...) Process

The *PCM_RECEIVE*(...) process defined in equations 39 and 40, allows to read a message from the channel. Once started with the *read_start*.*o*.*t* event it opens the valve enabling it to receive incoming poison. This is followed by checking whether the channel is already in a poisoned state. If that is the case, the valve will be closed again and the poison will be delivered to the calling process, using the *read_ex*.*o*.*t* channel, before terminating.

In case the channel is not poisonous, the process tries to read a message from the *transmit*.*o* channel. While waiting for the message to arrive it also waits for a potential poison to arrive over the *lp*2.*o* channel. Once the *transmit*.*o* channel delivered a message the valve is closed and the process delivers the received message using the *read_ack*.*o*.*t* channel. The process then terminates. If instead poison arrives from the *lp*2.*o* channel, the process closes the valve and delivers the poison using the *read_ex*.*o*.*t* channel. This is followed by terminating of the *PCM_RECEIVE*(...) process. This concludes the description of the *PCM_RECEIVE*(...) process. The interface of this process is shown in figure 15.

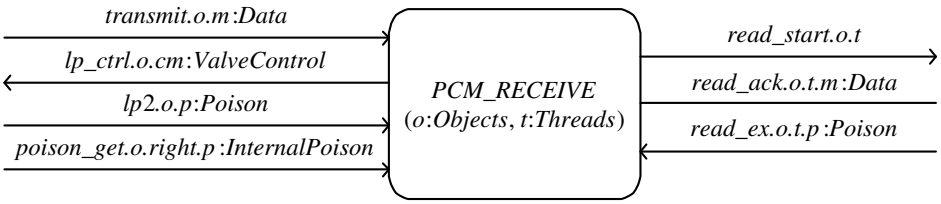


Figure 15. Interface of the *PCM_RECEIVE* process

$$
\begin{aligned}
&PCM\_RECEIVE(o : Objects, t : Threads) = \\
&\quad read\_start.o.t \rightarrow lp\_ctrl.o.open \rightarrow poison\_get.o.right?p \rightarrow \\
&\quad \left(\begin{array}{l}
\textbf{if } p = None \textbf{ then} \\
\quad \left(\begin{array}{l} transmit.o?m : Data \rightarrow lp\_ctrl.o.close \rightarrow \\ read\_ack.o.t!m \rightarrow SKIP \end{array} \right) \\
\quad \Box \left(\begin{array}{l} lp2.o?p \rightarrow lp\_ctrl.o.close \rightarrow \\ read\_ex.o.t!p \rightarrow SKIP \end{array} \right) \\
\textbf{else} \\
\quad lp\_ctrl.o.close \rightarrow \\
\quad read\_ex.o.t!p \rightarrow \\
\quad SKIP
\end{array} \right)
\end{aligned}
\tag{39}
$$

$$
\begin{aligned}
&\alpha PCM\_RECEIVE(o : Objects, t : Threads) = \\
&\quad \left\{ \begin{array}{l}
read\_start.o.t, read\_ack.o.t.m, ex.o.t.p, \\
lp\_ctrl.o.open, lp\_ctrl.o.close, lp2.o.p, \\
poison\_get.o.right.p, transmit.o.m \\
\quad \mid m \in Data, p \in InternalPoison
\end{array} \right\}
\end{aligned}
\tag{40}
$$

4.3.8. Preventing Concurrent Reading and Poisoning

The *PCM_RIGHT_CHOOSER*(...) process is used to prevent the reading process simultaneously performing a channel transaction and injecting poison into the channel. The definition of the process is given in equations 41 and 42. A graphical representation is given in figure 16.

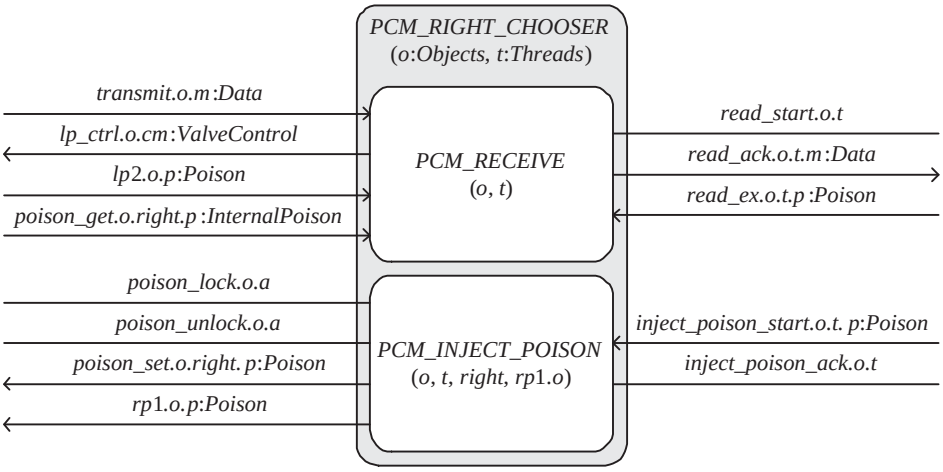


Figure 16. Interface of the *PCM_RIGHT_CHOOSER* process

$$
\begin{aligned}
&PCM\_RIGHT\_CHOOSER(o,t) = \\
&\quad (PCM\_RECEIVE(o,t) \Box PCM\_INJECT\_POISON(o,t,right,rp1.o)) ; \\
&\quad PCM\_RIGHT\_CHOOSER(o,t)
\end{aligned}
\tag{41}
$$

$$\begin{aligned} &\alpha PCM\_RIGHT\_CHOOSER(o : Objects, t : Threads) = \\ &\quad \alpha PCM\_RECEIVE(o,t) \cup \alpha PCM\_INJECT\_POISON(o,t,right,rp1.o) \end{aligned} \tag{42}$$

4.3.9. Reader Channel End

Similar to the writer channel end the reader channel end also needs to incorporate a $POISON\_VALVE(...)$ process for the $PCM\_RECEIVE(...)$ process, this is done in the $PCM\_RIGHT(...)$ process, which is a combination of the $PCM\_RIGHT\_CHOOSER(...)$ and $POISON\_VALVE(...)$ processes. The $PCM\_RIGHT(...)$ processes CSP model is given in equations 43 and 44. Figure 17 illustrates the interconnections of the combined processes.

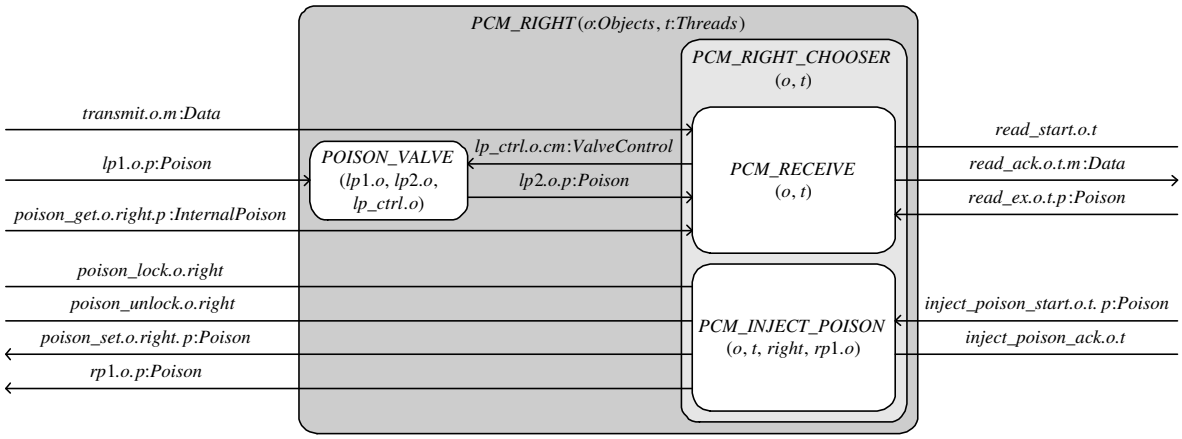


Figure 17. Interface of the *PCM_RIGHT* process

$$\begin{aligned} &PCM\_RIGHT(o : Objects, t : Threads) = \\ &\quad PCM\_RIGHT\_CHOOSER(o,t) \\ &\quad [\!| \alpha PCM\_RIGHT\_CHOOSER(o,t) \,|\, \alpha POISON\_VALVE(lp1.o, lp2.o, lp\_ctrl.o) |\!] \\ &\quad POISON\_VALVE(lp1.o, lp2.o, lp\_ctrl.o) \end{aligned} \tag{43}$$

$$\begin{aligned} &\alpha PCM\_RIGHT(o : Objects, t : Threads) = \\ &\quad \alpha PCM\_RIGHT\_CHOOSER(o,t) \cup \alpha POISON\_VALVE(lp1.o, lp2.o, lp\_ctrl.o) \end{aligned} \tag{44}$$

4.3.10. The $POISONABLE\_CHANNEL(...)$ Process

The Poisonable-Channel is a combination of the writer channel end (left side) and the reader channel end (right side). The channel ends are represented by the $PCM\_LEFT(...)$ and the $PCM\_RIGHT(...)$ processes. To avoid exposing internal events, the alphabet of the $POISONABLE\_CHANNEL(...)$ process is defined to be identical to the one defined for the interface *PC* defined in equation 11. The CSP model for the channel is given in equation 45, its alphabet in equation 46. The Poisonable-Channel structure is graphically represented in figure 18.

$$
\begin{aligned}
&\mathit{POISONABLE\_CHANNEL}(o : \mathit{Objects}, t_1 : \mathit{Threads}, t_2 : \mathit{Threads}) = \\
&\quad \left(\begin{array}{l} \mathit{PCM\_LEFT}(o, t_1) \\ \quad |[\, \alpha\mathit{PCM\_LEFT}(o, t_1) \mid \alpha\mathit{PCM\_RIGHT}(o, t_2) \,]| \\ \mathit{PCM\_RIGHT}(o, t_2)) \end{array} \right) \\
&\quad |[\, \alpha\mathit{PCM\_LEFT}(o, t_1) \cup \alpha\mathit{PCM\_RIGHT}(o, t_2) \mid \alpha\mathit{POISON}(o) \,]| \\
&\quad \mathit{POISON}(o)
\end{aligned} \tag{45}
$$

$$
\begin{aligned}
&\alpha\mathit{POISONABLE\_CHANNEL}(o : \mathit{Objects}, t_1 : \mathit{Threads}, t_2 : \mathit{Threads}) = \\
&\quad \alpha\mathit{PC}(o, t_1, t_2)
\end{aligned} \tag{46}
$$

4.4. Poison-Filtering-Channel

The Poison-Filtering-Channel is used to segment a process network into two parts, which can then be poisoned independently, by using *LocalPoison*. This is for instance a desired behaviour when having a design with fixed and a reconfigurable part, as is the case in a SDR Platform. There the signal processing module needs to be exchanged during runtime, without affecting the remaining platform. But even in segmented designs, there should be the ability to be terminated completely without caring about the segments. This was the reason for introducing GlobalPoison, which should be distributed into every corner of a process network.

4.4.1. The PFCM_INJECT_POISON(...) Process

The previous paragraph gave an indirect description of the behaviour of the Poison-Filtering-Channel. In short it should filter out any *LocalPoison* it gets injected, but let pass GlobalPoison. The process now needs to differentiate between *LocalPoison*, which is ignored, and *GlobalPoison* which will poison the channel. This results in a change of the *PCM_INJECT_POISON* process into the *PFCM_INJECT_POISON*(...) process given in equations 47 and 48. In this case the prefix *PFCM* indicates that this process is part of the Poison-Filtering-Channel Model.

$$
\begin{aligned}
&\mathit{PFCM\_INJECT\_POISON}(o : \mathit{Objects}, t : \mathit{Threads}, a : \mathit{PoisonAccess}, pc) = \\
&\quad \mathit{inject\_poison\_start}.o.t?p \rightarrow \\
&\quad \left(\begin{array}{l} \textbf{if } p = \mathit{GlobalPoison} \textbf{ then} \\ \quad \mathit{poison\_lock}.o.a \rightarrow \mathit{poison\_set}.o.a!p \rightarrow \mathit{poison\_unlock}.o.a \rightarrow \\ \quad\quad pc!p \rightarrow \mathit{inject\_poison\_ack}.o.t \rightarrow \mathit{SKIP} \\ \textbf{else} \\ \quad \mathit{inject\_poison\_ack}.o.t \rightarrow \mathit{SKIP} \end{array} \right)
\end{aligned} \tag{47}
$$

$$
\begin{aligned}
&\alpha\mathit{PFCM\_INJECT\_POISON}(o : \mathit{Objects}, t : \mathit{Threads}, a : \mathit{PoisonAccess}, pc) = \\
&\quad \left\{ \begin{array}{l} \mathit{inject\_poison\_start}.o.t.p, \mathit{inject\_poison\_ack}.o.t, \mathit{poison\_lock}.o.a, \\ \mathit{poison\_unlock}.o.a, \mathit{poison\_set}.o.a.p, pc.p \mid p \in \mathit{Poison} \end{array} \right\}
\end{aligned} \tag{48}
$$

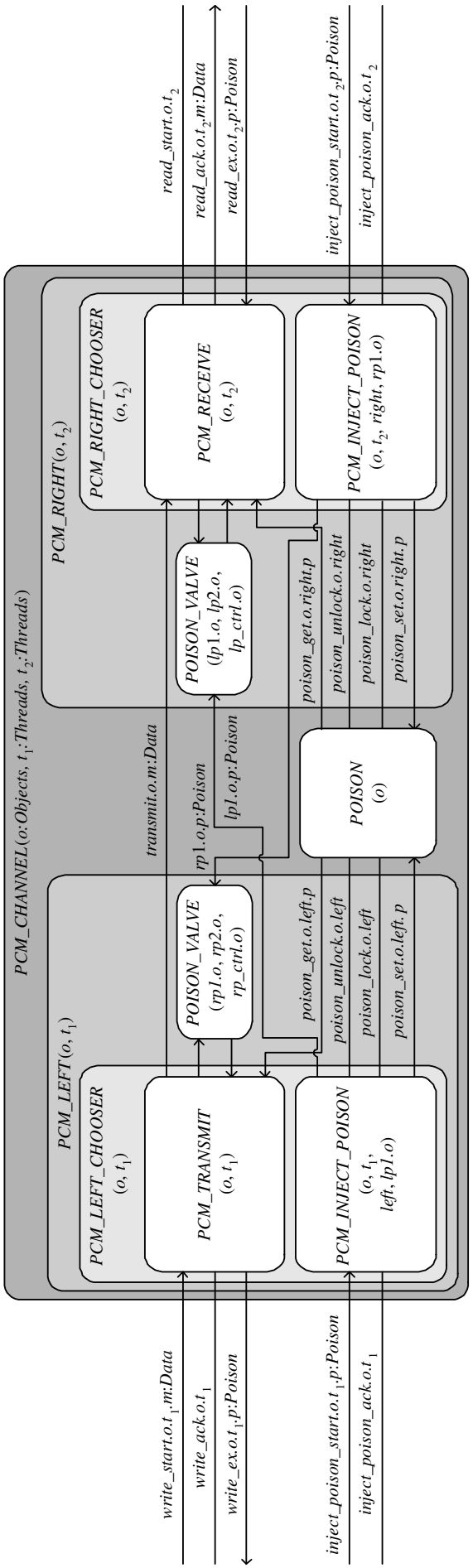


Figure 18.: Interface of the *PCM_CHANNEL* process

4.4.2. Writer Channel End

The writer channel end of the Poison-Filtering-Channel is represented by the process $PFCM\_LEFT(...)$, equations 49 and 50. This process offers two types of operation, either transmitting a message to the other channel end or injecting poison into the system.

$$\begin{aligned} &PFCM\_LEFT(o : Objects, t : Threads) = \\ &\quad PFCM\_LEFT\_CHOOSER(o,t) \\ &\quad |[\, \alpha PFCM\_LEFT\_CHOOSER(o,t) \mid \alpha POISON\_VALVE(rp1.o, rp2.o, rp\_ctrl.o) \,]| \\ &\quad POISON\_VALVE(rp1.o, rp2.o, rp\_ctrl.o) \end{aligned} \tag{49}$$

$$\begin{aligned} &\alpha PFCM\_LEFT(o : Objects, t : Threads) = \\ &\quad \alpha PFCM\_LEFT\_CHOOSER(o,t) \cup \alpha POISON\_VALVE(rp1.o, rp2.o, rp\_ctrl.o) \end{aligned} \tag{50}$$

The definition of the $PFCM\_LEFT\_CHOOSER(...)$ process is given in equations 51, 52.

$$\begin{aligned} &PFCM\_LEFT\_CHOOSER(o : Objects, t : Threads) = \\ &\quad (PCM\_TRANSMIT(o,t) \,\Box\, PFCM\_INJECT\_POISON(o,t,left,lp1.o)) \,; \\ &\quad PFCM\_LEFT\_CHOOSER(o,t)\big) \end{aligned} \tag{51}$$

$$\begin{aligned} &\alpha PFCM\_LEFT\_CHOOSER(o : Objects, t : Threads) = \\ &\quad \alpha PCM\_TRANSMIT(o,t) \cup \alpha PFCM\_INJECT\_POISON(o,t,left,lp1.o) \end{aligned} \tag{52}$$

4.4.3. Reader Channel End

The reader channel end offers a choice between reading from the channel and injecting poison into it. The reader side's CSP model is given in equations 53, 54.

$$\begin{aligned} &PFCM\_RIGHT(o : Objects, t : Threads) = \\ &\quad PFCM\_RIGHT\_CHOOSER(o,t) \\ &\quad |[\, \alpha PFCM\_RIGHT\_CHOOSER(o,t) \mid \alpha POISON\_VALVE(lp1.o, lp2.o, lp\_ctrl.o) \,]| \\ &\quad POISON\_VALVE(lp1.o, lp2.o, lp\_ctrl.o) \end{aligned} \tag{53}$$

$$\begin{aligned} &\alpha PCM\_RIGHT(o : Objects, t : Threads) = \\ &\quad \alpha PFCM\_RIGHT\_CHOOSER(o,t) \cup \alpha POISON\_VALVE(lp1.o, lp2.o, lp\_ctrl.o) \end{aligned} \tag{54}$$

The $PFCM\_RIGHT\_CHOOSER(...)$ processes definition is given in equations: 55, 56.

$$\begin{aligned} &PFCM\_RIGHT\_CHOOSER(o : Objects, t : Threads) = \\ &\quad (PCM\_RECEIVE(o,t) \,\Box\, PFCM\_INJECT\_POISON(o,t,right,rp1.o)) \,; \\ &\quad PFCM\_RIGHT\_CHOOSER(o,t) \end{aligned} \tag{55}$$

$$\begin{aligned} &\alpha PCM\_RIGHT\_CHOOSER(o : Objects, t : Threads) = \\ &\quad \alpha PCM\_RECEIVE(o,t) \cup \alpha PFCM\_INJECT\_POISON(o,t,right,rp1.o) \end{aligned} \tag{56}$$

4.4.4. The *POISON_FILTERING_CHANNEL*$(...)$ Process

The Poison-Filtering-Channel is split into the writer channel end (left side) and the reader channel end (right side). The channel ends are represented by the *PFCM_LEFT*$(...)$ and the *PFCM_RIGHT*$(...)$ processes. The CSP model for the channel is given in equation 57, its alphabet in equation 58.

$$\begin{aligned} &\mathit{POISON\_FILTERING\_CHANNEL}(o : \mathit{Objects}, t_1 : \mathit{Threads}, t_2 : \mathit{Threads}) = \\ &\left(\begin{array}{l} \mathit{PFCM\_LEFT}(o, t_1) \\ \quad [\![\, \alpha\mathit{PFCM\_LEFT}(o, t_1) \mid \alpha\mathit{PFCM\_RIGHT}(o, t_2) \,]\!] \\ \mathit{PFCM\_RIGHT}(o, t_2)) \end{array}\right) \\ &[\![\, \alpha\mathit{PFCM\_LEFT}(o, t_1) \cup \alpha\mathit{PFCM\_RIGHT}(o, t_2) \mid \alpha\mathit{POISON}(o) \,]\!] \\ &\mathit{POISON}(o) \end{aligned} \tag{57}$$

$$\begin{aligned} &\alpha\mathit{POISON\_FILTERING\_CHANNEL}(o : \mathit{Objects}, t_1 : \mathit{Threads}, t_2 : \mathit{Threads}) = \\ &\quad \alpha\mathit{PC}(o, t_1, t_2) \end{aligned} \tag{58}$$

4.5. Applying the Test-Bench

After developing the model for Poisonable-Channel and the Poison-Filtering-Channel it is necessary to see whether they comply with the test-bench defined in section 4.2.

4.5.1. Testing the Poisonable-Channel

Equation 59 shows the desired outcome when checking the Poisonable-Channel against the test-bench. All checks except the one against the *TB_LOCAL_POISONED_CHANNEL*$(...)$ process should succeed. Figure 19 shows a screen-shot of FDR with the results of the checks.

$$\begin{aligned} &\textbf{assert } \mathit{POISONABLE\_CHANNEL}(0,1,2) \\ &\quad \sqsubseteq_T \mathit{TB\_MESSAGE\_TRANSFER}(0,1,2,\mathit{TRUE}) \\ &\textbf{assert } \mathit{POISONABLE\_CHANNEL}(0,1,2) \\ &\quad \sqsubseteq_T \mathit{TB\_READER\_INJECTS\_POISON}(0,1,2,\mathit{LocalPoison}) \\ &\textbf{assert } \mathit{POISONABLE\_CHANNEL}(0,1,2) \\ &\quad \sqsubseteq_T \mathit{TB\_READER\_INJECTS\_POISON}(0,1,2,\mathit{GlobalPoison}) \\ &\textbf{assert } \mathit{POISONABLE\_CHANNEL}(0,1,2) \\ &\quad \sqsubseteq_T \mathit{TB\_WRITER\_INJECTS\_POISON}(0,1,2,\mathit{LocalPoison}) \\ &\textbf{assert } \mathit{POISONABLE\_CHANNEL}(0,1,2) \\ &\quad \sqsubseteq_T \mathit{TB\_WRITER\_INJECTS\_POISON}(0,1,2,\mathit{GlobalPoison}) \\ &\textbf{assert } \mathit{POISONABLE\_CHANNEL}(0,1,2) \\ &\quad \sqsubseteq_T \mathit{TB\_POISONED\_CHANNEL}(0,1,2,\mathit{LocalPoison}) \\ &\textbf{assert } \mathit{POISONABLE\_CHANNEL}(0,1,2) \\ &\quad \sqsubseteq_T \mathit{TB\_POISONED\_CHANNEL}(0,1,2,\mathit{GlobalPoison}) \\ &\textbf{assert } \mathit{POISONABLE\_CHANNEL}(0,1,2) \\ &\quad \not\sqsubseteq_T \mathit{TB\_LOCAL\_POISONED\_CHANNEL}(0,1,2) \end{aligned} \tag{59}$$

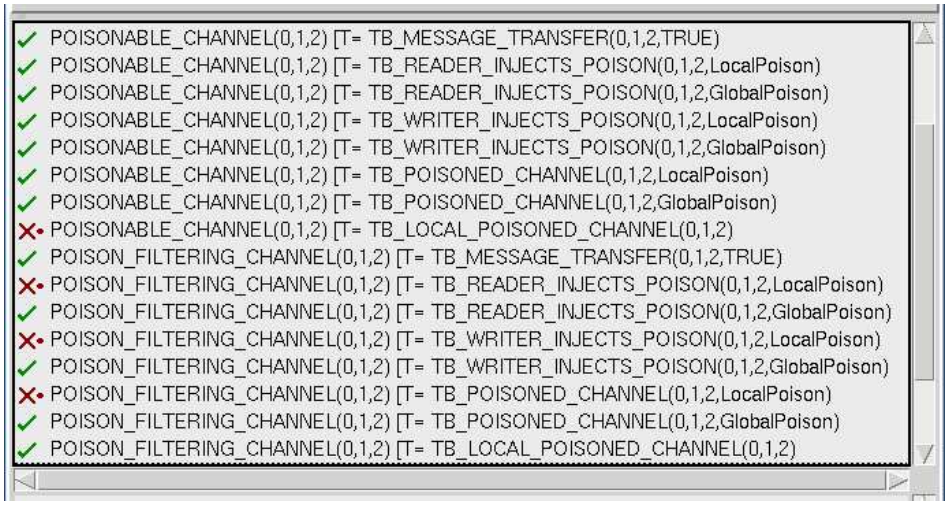


Figure 19. FDR Screen Shot showing the results of the refinement operations

4.5.2. Testing the Poison-Filtering-Channel

Testing the Poison-Filtering-Channel against the test-bench must result in the tests with *LocalPoison* to fail. This is caused by the fact that the Poison-Filtering-Channel is not transparent for *LocalPoison*. Which is of course its purpose. Instead the test against the $TB\_LOCAL\_POISONED\_CHANNEL(...)$ must succeed, to show that the Poison-Filtering-Channel is working properly. Checking was done using FDR and a screen-shot showing the results is given in figure 19.

$$
\begin{aligned}
&\textbf{assert } POISON\_FILTERING\_CHANNEL(0,1,2)\\
&\quad \sqsubseteq_T TB\_MESSAGE\_TRANSFER(0,1,2,TRUE)\\
&\textbf{assert } POISON\_FILTERING\_CHANNEL(0,1,2)\\
&\quad \not\sqsubseteq_T TB\_READER\_INJECTS\_POISON(0,1,2,LocalPoison)\\
&\textbf{assert } POISON\_FILTERING\_CHANNEL(0,1,2)\\
&\quad \sqsubseteq_T TB\_READER\_INJECTS\_POISON(0,1,2,GlobalPoison)\\
&\textbf{assert } POISON\_FILTERING\_CHANNEL(0,1,2)\\
&\quad \not\sqsubseteq_T TB\_WRITER\_INJECTS\_POISON(0,1,2,LocalPoison)\\
&\textbf{assert } POISON\_FILTERING\_CHANNEL(0,1,2)\\
&\quad \sqsubseteq_T TB\_WRITER\_INJECTS\_POISON(0,1,2,GlobalPoison)\\
&\textbf{assert } POISON\_FILTERING\_CHANNEL(0,1,2)\\
&\quad \not\sqsubseteq_T TB\_POISONED\_CHANNEL(0,1,2,LocalPoison)\\
&\textbf{assert } POISON\_FILTERING\_CHANNEL(0,1,2)\\
&\quad \sqsubseteq_T TB\_POISONED\_CHANNEL(0,1,2,GlobalPoison)\\
&\textbf{assert } POISON\_FILTERING\_CHANNEL(0,1,2)\\
&\quad \sqsubseteq_T TB\_LOCAL\_POISONED\_CHANNEL(0,1,2)
\end{aligned}
\tag{60}
$$

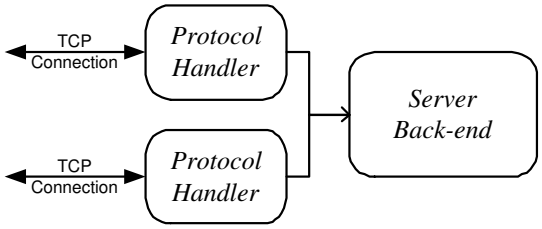


Figure 20. Process network of the Streamer

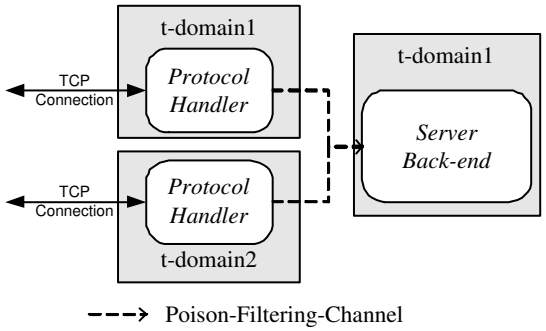


Figure 21. Termination Domains created by a Poison-Filtering-Channel

5. Applications / Examples

The example shown in this section was the initial motivation to look into the area of terminating sub networks.

The project we were working on was the development of a small server application to deliver signal data streams to multiple clients over the network. Similar to the SHOUTcast [13] or icecast [14] servers used for mp3 streaming. The functioning of this server was simple: a client should connect to a TCP socket provided by the service and then a small *Protocol Handler* process would be spun off, handling one client. The streaming of the data was handled by the *Server Back − end* process. The *Protocol Handler* was connected to the *Server Back − End* using an Any2OneChannel for data exchange. The resulting structure of the process network for two simultaneously connected clients is shown in figure 20. The spinning off of the *Protocol Handler* processes was no problem, due to the availability of the ProcessManager class in JCSP.

When a client disconnected from the server, the *Protocol Handler* process for this client detected this, and informed the *Server Back − end* process about this, to remove this client from all streams. After this the *Protocol Handler* becomes obsolete and should terminate. Naturally the *Server Back − end* should stay online, to be able to serve current and future clients. The solution during that time was to use the deprecated void ProcessManager.stop() method to terminate the *Protocol Handler* process. This was done despite the JCSP documentation warning about possible deadlocks. It worked for the streamer project but it was clear that a clean and safe solution had to be found. The result of this effort is presented in this paper.

So how to perform this termination in JCSP-Poison, without using the deprecated void ProcessManager.stop() method? It is actually quite simple. The Any2OneChannel used to connect the *Protocol Handler* and the *Server Back − end* has to be made a Poison-Filtering-Channel. Upon detection of a client disconnect the *Protocol Handler* process has to start to poison its channel ends using LocalPoison. In case the server back-end is terminated by the user, it must start to inject poison using GlobalPoison, which will then also affect the protocol handler process networks.

The resulting system is segmented in multiple termination domains, one for each protocol handler process network plus one for the server back-end, these of course only work if LocalPoison is injected. An additional termination domain exists when GlobalPoison is injected, then the complete network terminates. The different termination domains (t-domain) are illustrated in figure 21.

6. Conclusions

This paper showed a channel model supporting an easy way to terminate process networks. It furthermore showed how to enhance poisoning to support partial process network termination. This technique has further applications in the area of partial process network reconfiguration, where one process network is replaced by another. This is for instance necessary to create a CSP based Software Defined Radio.

The technique shown in this paper resolved some of the problems of the original graceful-termination technique proposed by P.H. Welch. Most important to mention is that now processes do not need to spin off black hole processes anymore to swallow any arriving messages. The possibility to poison data source processes, which only have outgoing channels, is a further advantage of the technique demonstrated here.

Other applications of the technique shown here, include the KCSP [15] environment, which currently provides complete process network termination. Unfortunately, this environment does not support the generation of exceptions, so the developer has to check the return value of each channel operation. But in the kernel environment it is even more important to perform proper clean up operations, than it is in user mode applications.

7. Further Work

Obviously, there are a lot of channel types not handled in this paper. To just name a few, there is still no model on how to deal with poison and the call channels provided by JCSP. Further investigations also include modelling the poisonable Any2One, One2Any and Any2Any channel types, to see whether they behave as expected.

The model shown in this paper was used to verify the correctness of the JCSP-Poison implementation model (which was not shown). This results in the shown model to be more complex than necessary, as it had to behave externally like a Java class. The next iteration of the shown model should be completely independent from JCSP and therefore become less complex. This should be done before modelling the other channel types available in JCSP.

In C++CSP Networked [16], channel ends can be restricted in their ability to poison a channel. Depending on the actual application of this feature, it is similar to use a Poison-Filtering-Channel and only use LocalPoison in the sub-network that should not affect the other networks. From a security standpoint, it is more powerful, as it allows to prevent a sub-network to poison its environment, by accidentally using a GlobalPoison instead of a LocalPoison. Therefore, the ability to prevent a channel end to inject poison into a channel should be added to the poisonable channels available in JCSP-Poison.

Partial process network reconfiguration in JCSP is also an area to look into, some ideas have already been drafted, but a proof of concept implementation is still missing.

A JCSP implementation of mobile processes [17], which are available for OCCAM already [18,19], will have to broadcast a message inside a sub-network. JCSP-Poison had similar problems when trying to poison a sub-network, so an extension of JCSP-Poison should be able to provide the desired sub-network broadcast functionality. Developing a concept to bring general message broadcasting, without using the barrier technique, to JCSP is a possible area to improveme JCSP-Poison.

Acknowledgements

The authors would like to thank P.H. Welch for providing the JCSP source code. Also appreciation to the reviewers for their careful reading.

References

[1] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

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

[3] Joseph Mitola. The software radio architecture. *IEEE Communications Magizine*, (5):26–38, May 1995.

[4] Joseph Mitola. Software radio architecture: A mathematical perspective. *IEEE Journal on Selected Areas in Communications*, 17(4):514–538, April 1999.

[5] Oliver Faust, Bernhard Sputh, and David Endler. Chaining Communications Algorithms with CSP. In Ian R. East, Prof David Duce, Mark Green, Jeremy M. R. Martin, and Peter H. Welch, editors, *Communicating Process Architectures 2004*, pages 325–338, 2004.

[6] P. H. Welch and P. D. Austin. Jcsp home page `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`.

[7] Cay S. Horstmann and Gary Cornell. *Core Java 2: Volume II-Advanced Features*, volume 2 of *The Sun Microsystems Press Java Series*. Sun Microsystems Press, A Prentice Hall Title, 901 San Antonio Road, Palo Alto, California, 94303-4900 U.S.A., 2002. ISBN: 0-13-092738-4.

[8] Peter H. Welch. Graceful termination – graceful resetting. In Andrè W. P. Bakkers, editor, *OUG-10: Applying Transputer Based Parallel Machines*, pages 310–317, 1989.

[9] P.H. Welch. Concurrency, exceptions and poison. Mailing List, September 2001. URL: `http://www.wotug.org/lists/occam/1076.shtml`.

[10] Neil C. Brown and Peter H. Welch. An Introduction to the Kent C++CSP Library. In Jan F. Broenink and Gerald H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 139–156, 2003.

[11] Formal Systems (Europe) Ltd. *FDR2 User Manual*, sixth edition, May 2003. URL: `http://www.fsel.com/fdr2_manual.html`.

[12] 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.

[13] nullsoft AOLmusic. SHOUTcast homepage. Internet. URL: `http://www.shoutcast.com`.

[14] Xiph.org Foundation. icecast homepage. Internet. URL: `http://www.icecast.org`.

[15] Bernhard Sputh. K-CSP Component Based Development of Kernel Extensions. In Ian R. East, Prof David Duce, Mark Green, Jeremy M. R. Martin, and Peter H. Welch, editors, *Communicating Process Architectures 2004*, pages 311–324, 2004.

[16] Neil C. Brown. C++CSP Networked. In Ian R. East, David Duce, Mark Green, Jeremy M. R. Martin, and Peter H. Welch, editors, *Communicating Process Architectures 2004*, pages 185–200, 2004.

[17] Fred Barnes and Peter H. Welch. Communicating Mobile Processes. In Ian R. East, David Duce, Mark Green, Jeremy M. R. Martin, and Peter H. Welch, editors, *Communicating Process Architectures 2004*, pages 201–218, 2004.

[18] Mario Schweigler, Frederick R. M. Barnes, and Peter H. Welch. Flexible, Transparent and Dynamic occam Networking With KRoC.net. In Jan F. Broenink and Gerald H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 199–224, 2003.

[19] Mario Schweigler. Adding Mobility to Networked Channel-Types. In Ian R. East, David Duce, Mark Green, Jeremy M. R. Martin, and Peter H. Welch, editors, *Communicating Process Architectures 2004*, pages 107–126, 2004.

Appendix. PoisonableOne2OneChannel Source Code

Listing 5 shows the actual implementation of the PoisonableOne2OneChannel in JCSP-Poison. The code is based upon the One2OneChannel provided by JCSP. The original listing is quite long, due to extensive commenting the code (originally this were seven pages). Some of the comments were removed where appropriate, but the comments concerned with the poisoning aspect of the system were left in. The code shown below is the basis for all poisonable channels. One of them, the Poison-Injector-Channel has to be able to differentiate between the two channel ends in terms of poison. This is the reason for the methods: isReaderPoisoned(), getReaderPoison(), isWriterPoisoned() and getWriterPoison(). In this channel no differentiation between reader and writer end poison is performed.

```
package jcsp.lang;

public class PoisonableOne2OneChannel
        extends PoisonableAltingChannelInput
        implements PoisonableChannelOutput {

    /**
     * The monitor synchronising reader and writer on this channel
     */
    protected Object rwMonitor = new Object();

    /**
     * The (invisible-to-users) buffer used to
     * store the data for the channel.
     */
    protected Object hold;

    /**
     * The synchronisation flag
     */
    protected boolean empty = true;

    /**
     * This flag indicates that the last transfer went OK. The
     * purpose is to not throw a PoisonException to the writer
     * side when the last transfer went OK, but the reader side
     * injected poison before the writer side finished processing
     * of the last write transfer.
     */
    protected boolean done = false;

    /**
     * The Alternative class that controls the selection
     */
    protected Alternative alt;

    /**
     * This member holds the poison
     * that was injected into the channel.
     */
    protected Poison poison = null;

    /**
     * This method is used whether the
     * reader side of the channel is poisoned.
     *
     * @return True if the reader side is poisoned, else false.
     */
    protected boolean isReaderPoisoned() {
        return (null != poison);
    }

    /**
     * This method is used to retrieve the type of
     * poison that the reader side is poisoned with.
     *
     * @return The poison that should be reported
     *         to the reader side.
     */
    protected Poison getReaderPoison() {
        return poison;
    }
```

```
/**
 * This method is used whether the
 * writer side of the channel is poisoned.
 *
 * @return True if the writer side is poisoned, else false.
 */
protected boolean isWriterPoisoned() {
    return (null != poison);
}

/**
 * This method is used to retrieve the type of poison that
 * the writer side is poisoned with.
 *
 * @return The poison that should be reported
 *         to the writer side.
 */
protected Poison getWriterPoison() {
    return poison;
}

/**
 * Reads an Object from the channel.
 *
 * @return the object read from the channel.
 *
 * @throws PoisonException in case the channel is poisoned.
 */
public Object read() throws PoisonException {
    synchronized (rwMonitor) {
        if (isReaderPoisoned()) {
            throw new PoisonException(getReaderPoison());
        }
        if (empty) {
            empty = false;
            try {
                rwMonitor.wait();
            } catch (InterruptedException e) {
                throw new ProcessInterruptedError(
                        "*** Thrown from Poisonable
                         One2OneChannel.read ()\n"
                        + e.toString());
            }
        } else {
            empty = true;
            rwMonitor.notify();
        }
        if (isReaderPoisoned()) {
            throw new PoisonException(getReaderPoison());
        } else {
            done = true;
            rwMonitor.notify();
            return hold;
        }
    }
}

/**
 * Writes an <TT>Object</TT> to the channel.
 * @param value the object to write to the channel.
 * @throws PoisonException in case the channel is poisoned.
 */
```

```
    public void write(Object value) throws PoisonException {
        synchronized (rwMonitor) {
            if (isWriterPoisoned()) {
                throw new PoisonException(getWriterPoison());
            }
            hold = value;
            if (empty) {
                empty = false;
                if (alt != null) {
                    alt.schedule();
                }
            } else {
                empty = true;
                rwMonitor.notify();
            }
            try {
                rwMonitor.wait();
            } catch (InterruptedException e) {
                throw new ProcessInterruptedError(
                        "*** Thrown from Poisnable
                         One2OneChannel.write (Object)\n"
                        + e.toString());
            }

            if (true == done) {
                done = false;
            } else {
                if (isWriterPoisoned()) {
                    hold = null;
                    throw new PoisonException(getWriterPoison());
                } else {
                    done = true;
                }
            }

        }
    }

    /**
     * Turns on Alternative selection for the channel.
     * @param alt the Alternative class which will control the selection
     * @return true if the channel has data that can be read
     *         or is poisoned, else false
     */
    boolean enable(Alternative alt) {
        synchronized (rwMonitor) {
            // When the channel is poisoned then make
            // it seem as if a message has arrived.
            if (isReaderPoisoned()) {
                return true;
            }
            if (empty) {
                this.alt = alt;
                return false;
            } else {
                return true;
            }
        }
    }
    /**
     * Turns off Alternative selection for the channel.
     * @return true if the channel has data that can be read
     *         or is poisoned, else false
     */
```

```
    boolean disable() {
        synchronized (rwMonitor) {
            alt = null;
            // Either data or Poison available for pickup.
            return (!empty || isReaderPoisoned());
        }
    }

    /**
     * Returns whether there is data pending on this channel.
     * @return true if the channel has data that can be read
     *         or is poisoned, else false
     */
    public boolean pending() {
        synchronized (rwMonitor) {
            // Data or Poison waiting for pickup
            return (!empty || isReaderPoisoned());
        }
    }

    /**
     * Injects poison into the channel.
     * @param poison the poison to inject into the channel.
     */
    public void injectPoison(Poison poison) {
        synchronized (rwMonitor) {
            // did get poison passed to the function?
            if (null == poison) {
                return;
            }
            // channel currently not poisoned
            if (null == this.poison) {
                this.poison = poison;
                // wake up possible sleeper
                rwMonitor.notifyAll();
                 // If alternation is used at the
                 //  reader side, alarm the reader.
                if (null != alt) {
                    alt.schedule();
                }
                // done
                return;
            }
            // is this Channel only poisoned with
            // LocalPoison?
            if (LocalPoison.class.isInstance(this.poison)) {
                // Global Poison overwrites LocalPoison
                if (GlobalPoison.class.isInstance(poison)) {
                    this.poison = poison;
                    // wake up possible sleeper
                    rwMonitor.notifyAll();
                    // If alternation is used at the
                    // reader side, alarm the reader.
                    if (null != alt) {
                        alt.schedule();
                    }
                    return;
                }
            }
        }
    }
}
```

Listing 5: Implementation of the PoisonableOne2OneChannel