Communicating Process Architectures 2013
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2013

Verifying the CPA Networking Stack using SPIN/Promela

Kevin CHALMERS and Jon KERRIDGE

Institute for Informatics & Digital Innovation, Edinburgh Napier University, UK

{K.Chalmers, J.Kerridge}@napier.ac.uk

Abstract. This paper presents a verification of the CPA Networking Stack, using the SPIN Model Checker. Our work shows that the system developed for general networking within CPA applications works under the conditions defined for it. The model itself focuses on ensuring deadlock freedom, and work still needs to be undertaken to verify expected behaviour of the architecture.

Keywords. distributed systems, model checking

Introduction

The net2 package of JCSP [1] introduced a general protocol and architecture that enables communication between CPA runtimes. For example, JCSP has been shown to interact with CSP .NET [2]. In this paper, a model of the general protocol and architecture is verified using the SPIN Model Checker [3]. By doing this work, it can be shown that the CPA Networking Stack is deadlock free, even under communication failure.

The rest of this paper is as follows. In Section 1 the motivation for this model is presented. In Section 2 a brief introduction to the SPIN Model Checker is given. Section 3 presents the architecture and operation of the CPA Networking Stack. Section 4 describes the model of the CPA Networking Stack built in SPIN and in Section 5 the results of checking this model is presented. Finally, in Section 6 conclusions and future work is discussed.

1. Motivation

One problem of the original networking implementation was poor error handling. Output net channels can fail due to link failure when exceptions are not passed to the application level. In the second networking implementation, output net channels are registered with the outgoing link. If the connection fails, the link signals each registered channel in turn. To determine if registering with the link avoids output channels hanging, a model of the architecture and protocol was developed with the SPIN Model Checker [3]. The model allows general verification of the architecture to check deadlock freedom and other properties of interest.

The CPA Networking stack itself was inspired in part by the T9000 virtual channel processor [4] which has been model checked in the past [5]. However, the CPA Networking Stack has fundamental differences (the use of channel mobility and a new protocol). Other developed CPA networking implementations, for example pony [6] and C++CSP Networked [7] have not been model checked at present.

2. SPIN/Promela

SPIN (Simple Promela INterpreter) [3] is a state space model checker allowing verification of a number of correctness requirements including assertion, deadlock, fairness and liveness checking of a model. The language of SPIN is Promela (PROcess MEta LAnguage) which has similar semantics to CSP (e.g. channels, processes, choice). The similarity enables composition of models from JCSP / .NET CSP source code into a SPIN model for verification.

To verify a model, SPIN converts Promela into C which is then compiled. The compiled application attempts to verify the model by exploring the state space which can involve the correctness requirements mentioned.

2.1. Using SPIN/Promela to Verify CPA Networking

Traditionally, CSP models are verified using FDR [8], or more recently PAT [9]. Although enabling refinement checking of the networking model would be advantageous, FDR does not currently support channel mobility.

Although it is possible to overcome this limitation [10], Promela permits channel mobility by passing channels as parameters in a message. The Promela channel is therefore similar to a channel in JCSP [11] / .NET CSP [12], although Promela does permit guarded operations on shared channels.

Because of the mobility of Promela channels the produced model is very similar to the implementation code. The processes in Section 3 that are also implemented in Section 4 are almost identical in JCSP and Promela code. Other processes described in Section 4 (e.g. InputNode, OutputNode, Network) are only necessary to verify the model.

3. CPA Networking Stack

In this section, an overview of the architecture and operation of the CPA Networking Stack will be provided. The information presented is important when the SPIN model is described in Section 4.

3.1. Architecture

Two architectural views of the networking stack are presented. The first provides a layered model allowing separation of functionality. The second examines the internal components of the layers, describing how they interact together to support the underlying distributed channel mechanism.

3.1.1. Layered Architecture

A layered view of CPA networking consists of four layers as represented in Figure 1:

Application Layer user level processes.

Event Layer networked synchronization primitives. Interfaces are provided to application layer processes, and the communication functionality of the components encapsulated.

Link Layer connections to other nodes, including receive (RX), transmit (TX), server, and manager processes.

Communication Layer the underlying communication mechanism that the CPA Networking stack sits upon.

On the left of Figure 1 the addressing mechanism is given, on the right the message types are given, and down the centre the interfaces between the layers is given. The interfaces are:

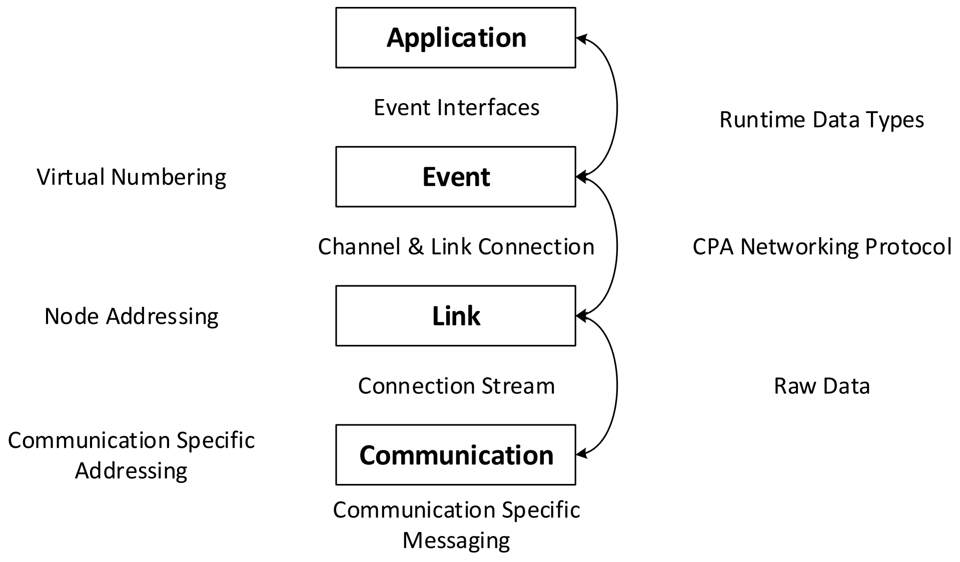

Figure 1. Layered Architecture of CPA Networking Stack.

Event Interfaces Interfaces with networked functionality added.

Channel & Link Connection Channels The crossbar allowing multiple links to communicate to multiple channel ends and multiple channel ends to communicate to multiple links. The crossbar is implemented using Any-2-One channels in both directions.

Connection Stream Links communicate with the communication mechanism using streams. These streams are communication specific.

Communication Specific Messaging The communication layer's messaging protocol (e.g. TCP/IP).

Addressing mechanisms between each layer are:

Virtual Numbering Number allocated for addressing and lookup purposes.

Node Addressing Each Node is uniquely identifiable to allow inter-Node connections. Link management relies on addressing to ensure that only one Link to a remote Node exists. An address takes the form $\langle \text{Protocol} \rangle \backslash\backslash \langle \text{Address} \rangle$. Protocol identifies the underlying communication mechanism (e.g. tcpip) and Address is the unique address of the Node based on the addressing mechanism of the communication mechanism.

Communication Specific Addressing Addressing mechanism enforced by the communication mechanism, for example $\langle \text{IP\_Address} \rangle : \langle \text{Port} \rangle$.

Each layer only understands certain message types. These are:

Runtime Data Types The Application Layer operates on data types dependent on the runtime used.

CPA Network Protocol The responsibility of the Event Layer is to convert outgoing messages into Network Protocol messages for communication via the Link Layer, and conversion of incoming Network Protocol messages from the Link Layer to communicate with the Application Layer. This protocol will be discussed further in Section 3.2

Raw Data Data leaving a Node is sent as bytes, which aids other platforms to interpret the incoming message. In particular protocol messages are transmitted as primitive data.

3.1.2. High Level Architecture

The individual components and how they are connected is presented in Figure 2.

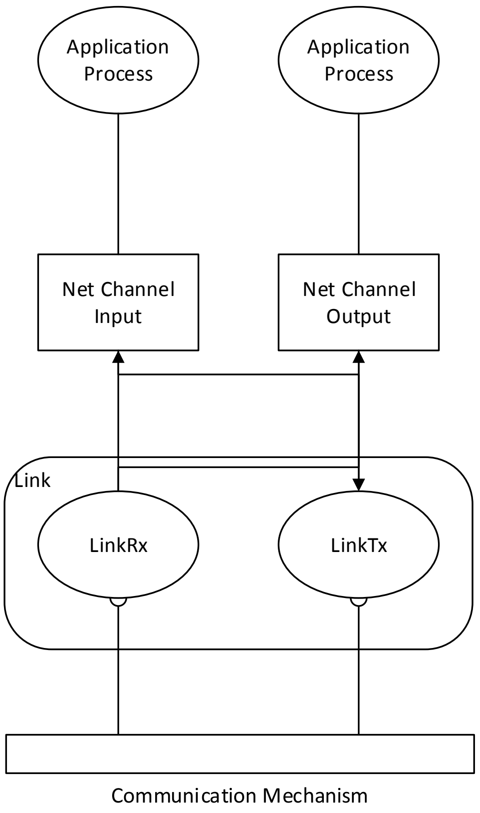

Figure 2. High-Level Architecture of CPA Networking Stack.

Link The Link component is responsible for connecting a node (a single running CPA application) to another node. The Link and its relevant subcomponents are designed to allow operation upon any communication mechanism if the necessary addressing and connection functionality is developed. The Link component has two sub-components which provide input and output operations between Nodes:

LinkTx The LinkTx process is responsible for transmitting messages to a remote node.

LinkRx The LinkRx process is responsible for receiving messages from a remote node. LinkRx interprets incoming messages and acts on the message type, accessing the destination channel if required.

Net Channel Output The networked output channel component provides the interface to the writing end of a networked channel, and hides the underlying interactions with the Link. The channel receiving messages from the Link is infinitely buffered.

Net Channel Input The networked input channel component provides the interface to the reading end of a networked channel.

3.2. Protocol

The primary goal of the CPA Networking protocol is that messages should be platform independent by using simple data primitives helps to achieve this goal.

3.2.1. Protocol Definition

The protocol is based on the original JCSP Networking implementation [13], the pony Framework [6] and C++CSP Networked [7]. The key feature is a virtual channel across a communication medium. Sent messages have a destination, and to permit synchronisation the message must be acknowledged, and therefore messages also have a source. Thus there are two attributes for a basic send message. The type of the message must also be included, providing a message triple. All required messages can be defined with a triple. There is also the optional data segment for data messages, providing the following message signature:

$$(\langle\text{type}\rangle, \langle\text{attr1}\rangle, \langle\text{attr2}\rangle, [\langle\text{data}\rangle])$$

Inclusion of data depends on $\langle\text{type}\rangle$, data size being sent as a header.
The basic channel message types are:

SEND Basic send message, requiring a source, destination, and data segment.
(`SEND`, $\langle\text{dest}\rangle, \langle\text{source}\rangle, \langle\text{data}\rangle$)

ACK `SEND` acknowledgement notifying the sender that a message has been read. Only the destination of the acknowledgement is required.
(`ACK`, $\langle\text{dest}\rangle$, null)

REJECT_CHANNEL Response to a `SEND` to a non-existent or destroyed channel. Only a destination is required.
(`REJECT_CHANNEL`, $\langle\text{dest}\rangle$, null)

POISON A `POISON` message requires a destination and a poison strength.
(`POISON`, $\langle\text{destination}\rangle, \langle\text{strength}\rangle$)

LINK_LOST When a Link fails, connected channels must be informed. `LINK_LOST` messages will never be transmitted between Nodes, but by a Link to local components. No extra information is required within the message.
(`LINK_LOST`, null, null)

ASYNC_SEND Unacknowledged `SEND` message. Asynchronous messages are used by server processes to allow typical distributed application request-respond interactions without blocking. `ASYNC_SEND` takes the same form as `SEND`.
(`ASYNC_SEND`, $\langle\text{dest}\rangle, \langle\text{source}\rangle, \langle\text{data}\rangle$)

3.3. Operation

In this section, the basic operations of CPA Networking are outlined. The methods to capture Link failure and data conversion are also covered. First a brief description of the new virtual channel is presented.

3.3.1. Virtual Channel

Figure 3 illustrates how components interact to form a networked channel. Messages between components illustrate the data that is being communicated.

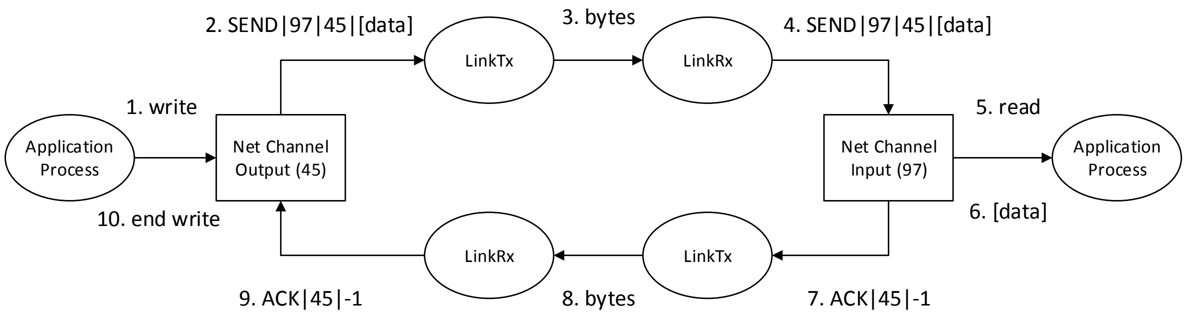

Figure 3. Virtual Networked Channel.

3.3.2. Basic SEND / ACK Operation

Figure 3 illustrates a standard read-write operation. A description of the steps involved is:

1. Application process writes to a networked output channel.
2. The networked output channel constructs a network message, setting the type as `SEND`, attr1 as the destination value (97) and attr2 as the source value (45). The sent data is attached to the network message and sent to the LinkTx. The networked output channel does not complete the write until the acknowledgement is received.
3. The LinkTx reads in the network message and writes the type (1) and two attributes (97 and 45) to the stream. The LinkTx examines the type of message, and as it is `SEND` there is a data portion. The LinkTx writes the size of the data and the bytes to the stream.
4. The receiving Node's LinkRx reads in the type attributes. The LinkRx process examines the message type, which is `SEND` and thus contains data. The size is read from the stream and used to read the required number of bytes. The LinkRx then retrieves the destination channel end from the channel index and checks its state. If the channel is in an `OK_INPUT` state the channel connecting to the partner LinkTx is added to the message, and the message sent to the networked input channel.
5. Application process reads the incoming message.
6. The networked input channel reads in the network message and checks the message type. As the type is `SEND` the message is to be delivered. The data is returned to the application process.
7. During step 6, a network message is created with the type `ACK`. attr1 is set to attr2 of the incoming message (the original source) and attr2 is set to -1. This message is written on the channel contained in the original message; the channel to the LinkTx process connected to the sending Node.
8. The LinkTx process reads the network message and writes the type (2) and the two attributes (45 and -1) to the stream. The LinkTx examines the type of the message, and as the type is `ACK` there is no data.
9. The original sending Node's LinkRx reads in the type and two attributes creating a network message from them. The LinkRx then examines the message type, and as it is a type that contains no data there is no need to read data from the stream. The LinkRx retrieves the channel from the channel manager and checks its state. If the channel is in an `OK_OUTPUT` state the network message is written to the networked output channel.
10. The networked output channel reads in the network message and checks the message type. As the type is `ACK` the write operation completes normally, freeing the application process.

The steps provided describe the operation under normal conditions. If the networked output channel is connected locally to a networked input channel, the same operations occur although at step 2 the message is sent directly to the local networked input channel component with the acknowledge channel of the networked output channel attached for direct acknowledgement.

As the architecture utilises I/O there is the possibility that erroneous behaviour can occur. The following sub-sections illustrate how this is handled.

3.3.3. SEND / REJECT operation

It is possible that erroneous message delivery can occur due to channel destruction or I/O operations. Message rejection is therefore implemented within the Link Layer. Figure 4 illustrates the component interactions that occur. The sequence of operations is:

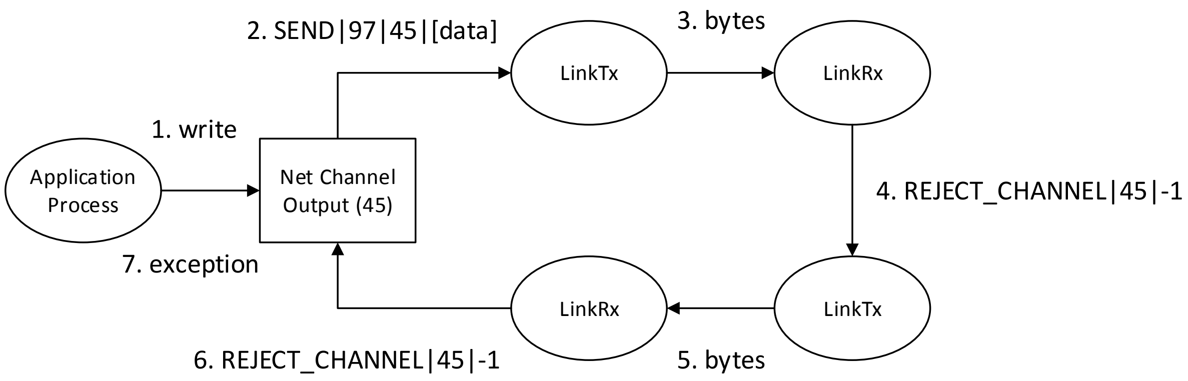

Figure 4. Reject Operation.

1. As normal operation
2. As normal operation
3. As normal operation
4. Initially this operation occurs as before. When the LinkRx attempts to retrieve channel 97 from the channel manager, the channel may not exist or its state may not be `OK_INPUT`. In either case, the LinkRx generates a network message and assigns the type `REJECT_CHANNEL`. attr1 is set to attr2 of the original message (45), and attr2 is not required. The network message is sent to the partner LinkTx.
5. As normal operation step 8, the LinkTx writes the message to the stream. There is no data segment.
6. The LinkRx reads in the type and two attributes. As the type contains no data segment, no data is read from the stream. The LinkRx then retrieves the necessary channel from the channel manager and checks the channel's state. If the channel is `OK_OUTPUT` the message is sent to the networked output channel.
7. The networked output channel reads in the message and checks the message type. As the message type is `REJECT_CHANNEL`, it is determined that the previous send was rejected. The networked output channel changes its state to `BROKEN` and removes itself from the channel manager. An exception is raised and causes the application process to continue but with an exception.

3.3.4. SEND / LINK_LOST

Another form of erroneous behaviour occurs when the connection to the node where the networked input channel resides fails. To overcome this, a networked output channel registers itself with a Link when it is created. As a networked output channel will only connect to one networked input channel, a Link can retain a set of all connected output channels. If the connection to the remote node is lost, the Link can inform all its registered channels by sending them a `LINK_LOST` message. Link failure may occur at any stage and therefore cannot easily be mapped into operational steps. There are two possibilities:

- Prior to a write operation, the Link to the remote node hosting the networked input channel fails, causing a `LINK\_LOST` message to be sent to the networked output channel on its acknowledgement channel. When write is called on the networked output channel, the acknowledgement channel is first checked for pending messages. As `LINK_LOST` will be present, the networked output channel can behave as if a message was rejected.
- After performing a write, but prior to receiving the `ACK`, the Link to the networked input channel fails. The Link informs all registered channel ends with a `LINK_LOST` message on their acknowledgement channels. The networked output channel will read

in this message, discover it is a LINK_LOST message and act as if the message was rejected.

By having all networked output channels register with Links, link failure can be transmitted as required. Networked input channels do not have this requirement as they may service multiple incoming connections. To avoid deadlock, the LinkTx remains active to black hole any outgoing messages. This is a current limitation of the CPA Networking Stack implementation - LinkTx processes do not die and remain active.

4. Building a Model of CPA Networking in Spin/Promela

4.1. Protocol Definition

SPIN uses the mtype keyword to define message types. From the discussion presented in Section 5.2, six message types within the protocol are relevant to channels. The ASYNC_SEND operation cannot be modelled as it can occur at any point during execution and requires no synchronisation between communicating components. This would increase the state space of the model beyond the capabilities of the model checker. An argument on its verification shall be presented in Section 5.

Discounting the ASYNC_SEND message, mtype is defined as follows:

```
mtype = {SEND, ACK, REJECT_CHANNEL, POISON, LINK_LOST};
```

4.2. Channel

A networked channel has a number of required definitions: the possible channel states, the data structure representing a channel, and the processes that represent networked input and output channels.

4.2.1. Channel States

Previous operational descriptions have mentioned channel states to determine how the Link Layer and Event Layer should behave. These state objects are shared between separate processes. The individual channel states are as follows:

INACTIVE initial channel state.

OK_INPUT a networked input channel willing to receive incoming messages.

OK_OUTPUT a networked output channel willing to send outgoing messages.

DESTROYED the channel end has been destroyed by an Application Layer process.

BROKEN a networked channel output end that has become broken due to some form of erroneous behaviour.

POISONED a channel end that has become poisoned, either by receiving a POISON message or by an Application Layer process invoking poison.

Figure 5 illustrates the transitions that occur between states within the channel.

4.2.2. Channel Data Structure

Each channel is provided with a data structure that contains the Virtual Channel Number (VCN), the state and the channel that the Link uses to communicate with the channel object. This is defined as follows:

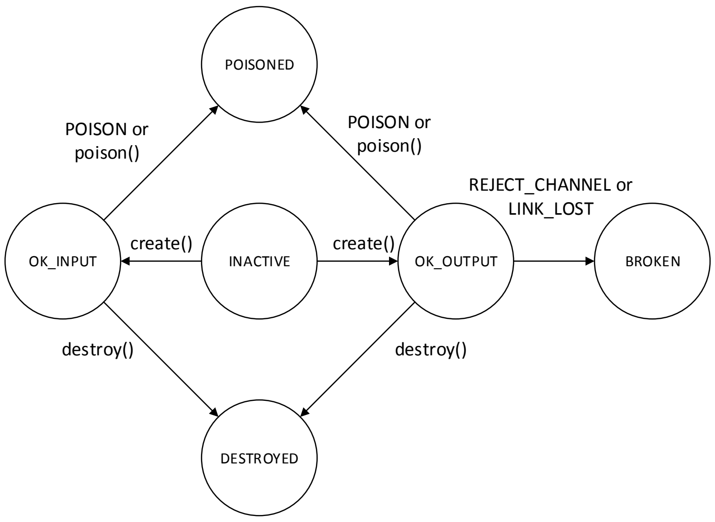

Figure 5. Channel States.

```
typedef CHANNEL_DATA
{
    byte vcn; byte state = INACTIVE; chan toChannel;
};
```

4.2.3. *Channel Process*

SPIN uses processes to represent components. A net channel process is given a `CHANNEL_DATA` structure to represent the channel, and an interface of channels that represent the possible calls that can be made on the channel. There are two channel types, NetChannelInput and NetChannelOutput. Figure 6 presents the NetChannelOutput process.

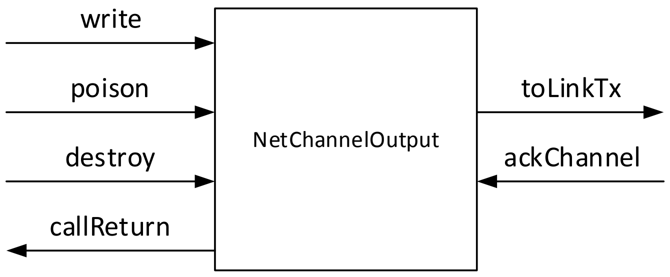

Figure 6. NetChannelOutput.

On the left of Figure 6 the interface channels are provided. Each channel represents the calling of a method on NetChannelOutput, except callReturn which is read to simulate the end of a call on the process.

On the right of Figure 6 are the channels connecting the channel process to the Link process. toLinkTx is a fixed channel that connects to the LinkTx where the input end of the virtual channel is connected. ackChannel is the channel coming from the Link, and is the channel defined in the `CHANNEL_DATA` type.

The Promela definition of the NetChannelOutput interface is as follows:

```
typedef OUTPUT_CHANNEL_INTERFACE
{
    chan write = [0] of { bool };
    chan poison = [0] of { bool };
    chan destroy = [0] of { bool };
    chan callReturn = [0] of { bit };
};
```

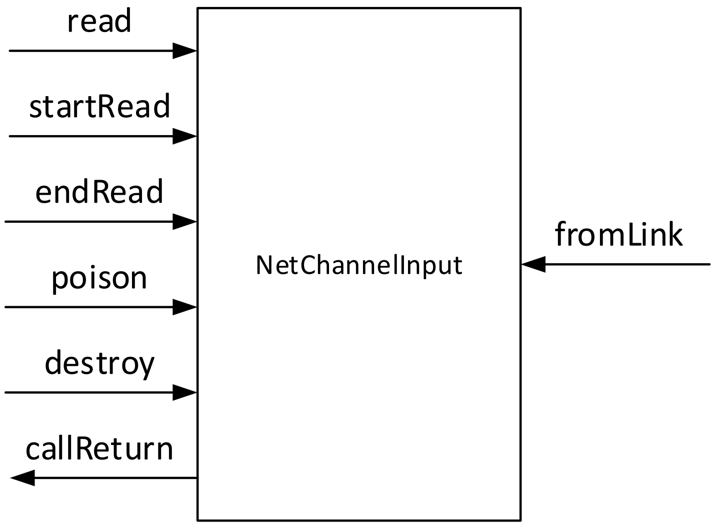

Figure 7. NetChannelInput.

Figure 7 presents the NetChannelInput process. The method interface is on the left, and includes extended rendezvous and poison operations. For completeness these operations are added to the SPIN model.

The NetChannelInput process has only one connection to the Link processes, the fromLink channel. This channel is as declared in the `CHANNEL_DATA` type. When a Link sends the NetChannelInput a message, it also sends the channel to send the response back to the Link. This is where channel mobility is required.

The Promela definition of the NetChannelInput interface is as follows:

```
typedef INPUT_CHANNEL_INTERFACE
{
    chan read = [0] of { bool };
    chan startRead = [0] of { bool };
    chan endRead = [0] of { bool };
    chan poison = [0] of { bool };
    chan destroy = [0] of { bool };
    chan callReturn = [0] of { bit };
};
```

4.3. Link Process

Link contains two processes: LinkTx and LinkRx. LinkTx receives messages from the channel processes and sends them to the remote LinkRx. It has two channels (one in and one out). input receives messages from the channel processes. txStream represents the connection to the remote LinkRx process.

LinkRx receives messages from a remote LinkTx and sends them to the correct channel. It also has two channels. rxStream represents the incoming stream from the remote LinkTx. toLinkTx connects to the complement LinkTx, and is used to send messages directly to the LinkTx and to attach to incoming messages to allow a subsequent acknowledgement to be sent directly to the LinkTx.

A Link represents a connection to another node, and is composed of a LinkTx and a LinkRx. Figure 8 represents the Link process.

As the LinkRx contains the protocol behaviour, the full Promela code listing is provided in Appendix A. The majority of the networking stack behaviour is contained in this process.

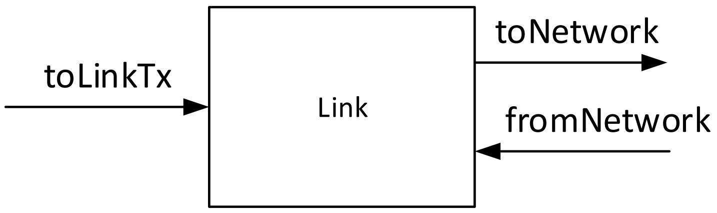

Figure 8. Link Process.

4.4. Application Processes

There are two types of application process: an outputting application and an inputting application. These processes operate on the complement end interface channels that connect to a NetChannelInput or NetChannelOutput. The application process chooses non-deterministically to write to one of the method call channels and then reads from the callReturn channel, thus waiting for the operation to complete. callReturn returns either 0 or 1 to represent either an `EXCEPTION` or an OK return message. If an `EXCEPTION` is returned, then the application process terminates.

4.5. Node Process

Within the model, two node types are defined: InputNode and OutputNode. An InputNode starts a number of Receiver processes with relevant NetChannelInput processes. An OutputNode starts a number of Sender processes and relevant NetChannelOutput processes. Figure 9 presents the InputNode process. The connection between the Link process(es) and the NetChannelInput process(es) is shown, although this is dynamic.

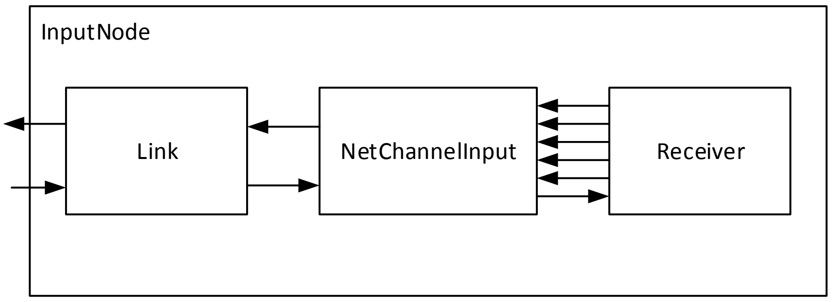

Figure 9. InputNode.

Figure 10 illustrates the OutputNode process. In this circumstance, the connection between the NetChannelOutput and Link is static.

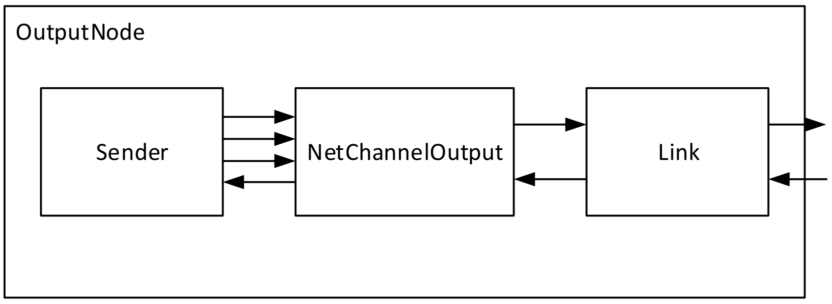

Figure 10. OutputNode.

For both Nodes, the toNetwork and fromNetwork channels represent the txStream and rxStream across the network connecting the two remote Nodes. A process is also added that allows simulation of the network connection itself.

4.6. Network Process

To simulate network failure, a simple process to represent the network is added to the model. The process non-deterministically chooses to either send a message from the OutputNode to the InputNode, from the InputNode to the OutputNode, or fail and break the connection. In the latter case, a LINK_LOST signal is sent to the two corresponding LinkRx processes, and a flag is set which the LinkTx processes check to determine if they should fail. In reality the latter occurrence is detected when the LinkTx process tries to write to a closed stream. The setting of a flag achieves the same outcome.

Although I/O can fail in different manners, the CPA Networking Stack expects the underlying I/O mechanism to handle this and simply inform the Link processes of a closed connection. As such, the model does not need to check behaviour of a partial communication being recieved - either the whole message is available or it is not.

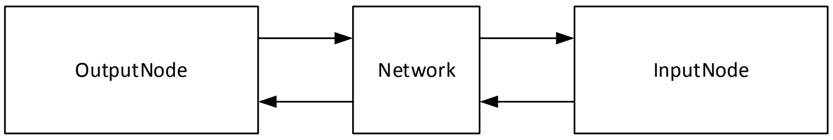

Figure 11. SPIN Model of CPA Networking.

Figure 11 presents the overall SPIN model developed for CPA Networking. The two nodes are connected via the Network process.

The Promela definition of the Network process is as follows:

```
proctype Network(chan in0; chan in1; chan out0; chan out1)
{
    mtype type;
    byte attr1;
    byte attr2;
end_network:
    do
    :: atomic
       {
            in0 ? type, attr1, attr2 -> out1 ! type, attr1, attr2
       }
    :: atomic
       {
            in1 ? type, attr1, attr2 -> out0 ! type, attr1, attr2
       }
    /* Non determistically choose to break link */
    :: atomic
       {
            linkLost = true ->
            out0 ! LINK_LOST(-1, -1);
            out1 ! LINK_LOST(-1, -1);
       }
       break;
    od
}
```

4.7. Global Values

There are a number of global values within the model, and these are summarised below:

NUMBER_INPUTS the total number of input channels within the model.

NUMBER_OUTPUTS the number of output channels connected to a single input channel

TOTAL_OUTPUTS the total number of output channels NUMBER_OUTPUTS * NUMBER_INPUTS

BUFFER_SIZE the size of the buffer to the channel processes. This is used to simulate the infinite buffer within the actual application. For the application to operate, BUFFER_SIZE should equal NUMBER_OUTPUTS. This value is manipulated to verify this assumption

CHANNEL_ARRAY a type declaration for the array of channel ends on a particular node CHANNEL_DATA channels[TOTAL_OUTPUTS]. SPIN does not permit arrays to be passed as parameters into processes, therefore this must be declared globally. For channels above NUMBER_INPUTS on the InputNode, the channel state is set to INACTIVE.

chans all the channels within the model. As CHANNEL_ARRAY cannot be passed as a parameter to an individual process, this is declared globally.

linkLost the flag used to indicate Link failure. This is initially set to false.

```
byte linkLost = false;

typedef CHANNEL_ARRAY
{
    CHANNEL_DATA channels[TOTAL_OUTPUTS];
};

CHANNEL_ARRAY chans[2];
```

5. Results

5.1. Basic Verification

Simple verification can be carried on a model comprising of a single NetChannelOutput connected to a NetChannelInput with BUFFER_SIZE = 1. This is the default assumption that for every connected output channel end to an input channel end, there is required a single place in the buffer to avoid deadlock. This assumption is required for the worst case scenario - the input end does not read while all output channels attempt to write, and therefore require a buffer space for their message to be stored. When checked with SPIN, the model is verified with no deadlock errors. This is enough to reasonably assume that having the Link processes inform the relevant output channels of connection failure overcomes the deadlock problem in the original JCSP Networking implementation.

The model also allows some indication of other problems in the original JCSP Networking implementation. In the new implementation, the LinkRx process retrieves a channel from the channel manager and locks the state object of the channel before checking said state. Thereby, LinkRx is the only process acting on the channel state at any one time. This allows various behaviours to occur based on the state of the channel object. This feature was added to the implementation of JCSP Networking when the model originally pointed out deadlock due to this occurrence not being taken into consideration. As the channel object can change state based on certain calls (poison, destroy), this would have caused inconsistent behaviour within the implementation. The original implementation used no such state variable, and LinkRx would send a message to a channel object based purely on availability within the channel manager. The channel manager would only return the connecting output channel to the networked channel object. When a channel object was destroyed, it was removed from the channel manager prior to any clean up operations (rejection of pending messages). Therefore, a channel either existed within the channel manager or it did not. There were no other possible states as no common protected state value was exposed. This meant that much of the behaviour required for more advanced functionality (poison, mobility, barriers) was not possible as there was no method to expose these states without reimplementation of the underlying mechanisms of JCSP Networking. As the new implementation exposes these properties, this problem has been overcome.

5.2. Advanced Verification

The simple verified model does not allow analysis of the assumption that the a networked input channel NetChannelInput requires exactly one buffer space for each connected NetChannelOutput. With manipulation of the BUFFER_SIZE value, this can be analysed to provide a stronger insight into this assumption. Table 1 presents results from different verification scenarios. To enable verification of the model, the option within SPIN to use minimal automata to search is activated.

Table 1. SPIN Verification Results.

NUMBER_OUTPUTS	**1**	**2**	**3**	**4**
BUFFER_SIZE				
0	FAIL	FAIL	FAIL	FAIL
1	PASS	FAIL	FAIL	FAIL
2	PASS	PASS	FAIL	FAIL
3	PASS	PASS	PASS	FAIL
4	PASS	PASS	PASS	PASS

Table 1 illustrates that a NetChannelInput requires one buffer space for each connected NetChannelOutput for connected NetChannelOutputs less than four. The number of states does not increase when the buffer size is increased beyond the required buffer size, except when a single NetChannelOutput to NetChannelInput has the buffer increased from 1 to 2, although search depth does not increase. The reason for the reduction in state space could be the usage of the minimal automata search option within SPIN, or that the NetChannelOutput requires less state space in conjunction with the Link processes at BUFFER_SIZE = 2. The NetChannelOutput also utilises the same size buffer as the NetChannelInput in the model, and this could have an effect in total required states.

6. Conclusions and Future Work

This paper has shown that the CPA Networking Stack is deadlock free even under network failure, a major issue in the original JCSP Networking implementation. The work has provided an overview of the model built using SPIN, and shown the relation between connected networked output channels and the size of buffer required to service them.

Future work required in the formal verification of the CPA Networking Stack involves refinement checking to determine if the general behaviour of the CPA Networking Stack is correct. The model will have to show that a virtual networked communication behaves similarly to a standard channel communication.

References

[1] Kevin Chalmers, Jon Kerridge, and Imed Romdhani. A Critique of JCSP Networking. In Peter H. Welch, S. Stepney, F.A.C Polack, Frederick R. M. Barnes, Alistair A. McEwan, G. S. Stiles, Jan F. Broenink, and Adam T. Sampson, editors, *Communicating Process Architectures 2008*, pages 271–291, sep 2008.

[2] Kevin Chalmers. Performance of the Distributed CPA Protocol and Architecture on Traditional Networks. In Peter H. Welch, Adam T. Sampson, Jan Baekgaard Pedersen, Jon Kerridge, Jan F. Broenink, and Frederick R. M. Barnes, editors, *Communicating Process Architectures 2011*, pages 227–242, jun 2011.

[3] G.J. Holzmann. The model checker SPIN. *Software Engineering, IEEE Transactions on*, 23(5):279–295, 1997.

[4] D. May, R. Shepherd, and P. Thompson. The T9000 transputer. In *Computer Design: VLSI in Computers and Processors, 1992. ICCD '92. Proceedings, IEEE 1992 International Conference on*, pages 209–212, 1992.

[5] Geoff Barrett. Model checking in practice: the T9000 virtual channel processor. *Software Engineering, IEEE Transactions on*, 21(2):69–78, 1995.

[6] Mario Schweigler and Adam T. Sampson. pony - The occam-pi Network Environment. In Peter H. Welch, Jon Kerridge, and Frederick R. M. Barnes, editors, *Communicating Process Architectures 2006*, pages 77–108, sep 2006.

[7] Neil C.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, sep 2004.

[8] Formal Systems (Europe) Ltd. *FDR: User Manuel and Tutorial, version 2.82*, 2005.

[9] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. PAT: Towards flexible verification under fairness. volume 5643 of *Lecture Notes in Computer Science*, pages 709–714. Springer, 2009.

[10] Peter H. Welch and Frederick R. M. Barnes. A CSP Model for Mobile Channels. In Peter H. Welch, S. Stepney, F.A.C Polack, Frederick R. M. Barnes, Alistair A. McEwan, G. S. Stiles, Jan F. Broenink, and Adam T. Sampson, editors, *Communicating Process Architectures 2008*, pages 17–33, sep 2008.

[11] P. H. Welch. Process oriented design for Java: concurrency for all. pages 51–57. CSREA Press, 2000.

[12] Kevin Chalmers and Sarah Clayton. CSP for .NET Based on JCSP. In Peter H. Welch, Jon Kerridge, and Frederick R. M. Barnes, editors, *Communicating Process Architectures 2006*, pages 59–76, sep 2006.

[13] Peter H. Welch, Jo R. Aldous, and Jon Foster. CSP networking for Java (JCSP.net). In *Proceedings of the International Conference on Computational Science-Part II*, ICCS '02, pages 695–708, London, UK, UK, 2002. Springer-Verlag.

A. Promela Code for LinkRx

```
proctype LinkRx(chan toTxProcess; chan rxStream; bit nodeNumber)
{
    /* Attributes read in with incoming message */
    byte attr1;
    byte attr2;

    /* This process reads an incoming message from the message and
       processes it.  Generally the message
       is forwarded onto the correct destination, although erroneous
          behaviour must be dealt with */
    /* Valid end state.  Waiting for input from the network */
end_lrx:
    do
    /* SEND received.  */
    :: atomic
      {
            rxStream ? SEND(attr1, attr2) ->
            /* First check if the message is going to a valid channel
               */
            if
            /* Destination channel is outside range.  Reject message
               */
            :: (attr1 > TOTAL_OUTPUTS) -> toTxProcess ! REJECT_CHANNEL
               (attr2, -1)
            :: else ->
                 /* Message is for a valid channel.  Check channel
                    state and deal with accordingly */
```

```
              if
              :: (chans[nodeNumber].channels[attr1].state ==
                 OK_INPUT) ->
                  /* Channel is OK to receive messages.  Forward the
                       message onto the channel process */
                  chans[nodeNumber].channels[attr1].toChannel ! SEND
                     (attr2, toTxProcess)
              :: (chans[nodeNumber].channels[attr1].state ==
                 POISONED) ->
                  /* Channel has been poisoned.  Propogate the
                     poison back to the writer */
                  toTxProcess ! POISON(attr2, 0)
              :: (chans[nodeNumber].channels[attr1].state ==
                 DESTROYED) ->
                  /* Channel has been destroyed.  Reject the message
                       */
                  toTxProcess ! REJECT_CHANNEL(attr2, 0)
              :: (chans[nodeNumber].channels[attr1].state == BROKEN)
                  ->
                  /* Channel is broken.  This should only happen
                     during Link failure, but just in case reject */
                  toTxProcess ! REJECT_CHANNEL(attr2, 0)
              :: else ->
                  /* Channel is in some other state.  This could be
                     a channel trying to send to an output
                     or some other problem.  We reject the message
                        in this instance and continue */
                  toTxProcess ! REJECT_CHANNEL(attr2, 0)
              fi
          fi
    }
  /* Acknowledgement operation */
  :: atomic
    {
      rxStream ? ACK(attr1, attr2) ->
          /* First check if the message is going to a valid channel
             */
          if
          /* Destination channel is outside range.  Ignore message
             */
          :: (attr1 > TOTAL_OUTPUTS) -> skip
          :: else ->
              /* Message is for a valid channel.  Check channel
                 state and deal with accordingly */
              if
              :: (chans[nodeNumber].channels[attr1].state ==
                 OK_OUTPUT) ->
                  /* ACK being sent to an output channel.  Forward
                     the message onto the channel process */
                  chans[nodeNumber].channels[attr1].toChannel ! ACK
              /* In all other cases, we drop the message.  The
                 message has been sent to a channel that was
                 not in a state to accept it. */
              :: else -> skip
              fi
          fi
    }
  /* Reject channel message received */
  :: atomic
    {
      rxStream ? REJECT_CHANNEL(attr1, attr2) ->
          /* First check if the message is going to a valid channel
             */
          if
          /* Destination channel is outside range.  No point in
             rejecting (we could end up with a continuous
             cycle of rejects). Simply ignore the message. */
          :: (attr1 > TOTAL_OUTPUTS) -> skip
          :: else ->
              /* Message is for a valid Channel.  Check channel
                 state and deal with accordingly */
              if
              :: (chans[nodeNumber].channels[attr1].state ==
                 OK_OUTPUT) ->
                  /* Channel can accept the reject message.  Pass
                     onto the channel process */
                      chans[nodeNumber].channels[attr1].toChannel !
                         REJECT_CHANNEL
              /* In all other cases ignore the message.  The channel
                  is in no state to receive it */
              :: else -> skip
              fi
          fi
    }
  /* Poison message received */
  :: atomic
    {
      rxStream ? POISON(attr1, attr2) ->
          /* First check if the message is going to a valid channel
             */
          if
          /* Destination channel is outside range.  No point in
             rejecting (we could end up with a continuous
             cycle of rejects). Simply ignore the message. */
          :: (attr1 > TOTAL_OUTPUTS) -> skip
          :: else ->
              /* Message is for a valid Channel.  Check channel
                 state and deal with accordingly */
              if
              :: (chans[nodeNumber].channels[attr1].state ==
                 OK_OUTPUT) ->
                  /* Channel is an output.  Simply send POISON to it
                     . */
                  chans[nodeNumber].channels[attr1].toChannel !
                     POISON
              :: (chans[nodeNumber].channels[attr1].state ==
                 OK_INPUT) ->
                  /* Channel is an input.  Simply send POISON to it
                     */
                  chans[nodeNumber].channels[attr1].toChannel !
                     POISON(attr1, attr2)
              /* In all other cases we ignore the poison.  Either
                 the channel is poisoned, and in the model
                 nothing else needs to be done (in the
                    implementation we increase the poison strength
                    if
                 necessary), or it is destroyed or broken, which is
                    considered to be greater than poison */
                  :: else -> skip
                  fi
              fi
        }
      /* Link lost received */
      :: rxStream ? LINK_LOST(attr1, attr2) ->
          atomic
          {
              /* Inform all output ends */
              byte idx = 0;
              do
              :: (idx < TOTAL_OUTPUTS) ->
                  if
                  :: (chans[nodeNumber].channels[idx].state == OK_OUTPUT
                     ) ->
                      chans[nodeNumber].channels[idx].toChannel !
                         LINK_LOST
                  :: else -> skip
                  fi;
                  idx = idx + 1;
              :: else -> break
              od;
          }
          break;
      od;
  }
```