COMMS/CPN: A Communication Infrastructure for External Communication with DESIGN/CPN

Guy Gallasch and Lars Michael Kristensen

Computer Systems Engineering Centre
School of Electrical and Information Engineering
University of South Australia
Mawson Lakes Campus, SA 5095, AUSTRALIA
Email: {galgy002@students.unisa.edu.au,lars.kristensen@unisa.edu.au}

Abstract. In this paper the development of COMMS/CPN is presented. COMMS/CPN is a Standard ML library that augments DESIGN/CPN with the necessary infrastructure to establish communication between CPN models and external processes. COMMS/CPN is potentially beneficial in a number of areas such as allowing external visualisations of simulations, providing CPN models with their own Graphical User Interface, and allowing CPN models to interact with the physical environment. COMMS/CPN has been successfully applied for providing external visualisation of the simulation of a CPN model within the area of avionics mission systems.

1 Introduction

Coloured Petri Nets (CPNs) [11,13], when constructed and simulated using the DESIGN/CPN tool [21], are restricted in their ability to interact with external processes. Extending the DESIGN/CPN tool by providing a communication infrastructure allows communication to be established between CPN models and external processes. The COMMS/CPN library [6] presented in this paper has been developed to extend DESIGN/CPN with such external communication facilities.

The motivation behind developing external communication facilities comes from the desire to visualise the simulation of CPN models. As demonstrated in [2, 23], it is often beneficial to extend CPN models with application specific graphics. The behavior of the system under consideration can be visualised using different kinds of graphical feedback. This provides a view of system behavior useful for system developers and analysts, and can also be useful for conveying knowledge and results about CPN models to people not familiar with CPN modelling and analysis.

Currently, DESIGN/CPN provides visualisation capabilities in two forms. The first is the token game which displays the simulation of a CPN model in a very detailed fashion. The second is by using high-level application specific graphics that can be added on top of a CPN model. MIMIC/CPN [1] and the Message Sequence Chart library [18] provide such high-level graphics. However these methods are not always satisfactory. The token game is often too detailed, and the application specific graphics are sometimes limited in capability and capacity, and are tied to the Graphical User Interface (GUI) of DESIGN/CPN. It is therefore of interest to conduct visualisation using an external application. External applications can be developed with greater graphical capabilities than those of DESIGN/CPN, and there is the potential to execute the external application on a remote machine.

The COMMS/CPN library has been developed to allow communication between DESIGN/CPN and external processes via TCP/IP [4]. The main benefits that the COMMS/CPN infrastructure will provide in the context of visualisation are:

– The infrastructure makes it possible to visualise the behavior of CPN models and control their simulation independently of the Design/CPN GUI.
– The infrastructure provides flexibility, since other graphical libraries and packages are likely to provide better support for visualisation than Design/CPN.
– The infrastructure makes it possible to do the visualisation on remote machines provided they support TCP/IP communication.

It should be stressed that Comms/CPN is not limited to use in external visualisation. Comms/CPN has the potential to be beneficial in many other areas. As an example, it could be used to provide CPN models with their own GUI. The Design/CPN simulator is built on the Standard ML (SML) [20,27] compiler and Comms/CPN is also implemented in SML. This means that the Design/CPN GUI could be separated from the simulator, and a GUI specific to the CPN model could be used instead. An example where this may be useful is when applying CPN models in decision making processes, as shown in [15]. Using Comms/CPN, it is also conceivable that CPN models could interact with the physical environment. Examples include temperature and light sensors, keypads, and displays (although such experiments have not yet been conducted). Situations may also arise where computationally expensive algorithms and procedures are needed within Design/CPN. With Comms/CPN, these can be implemented and executed on remote machines, and the results can be sent back to the CPN model. An example of this can be found in [17] where the condensed state space tool of Design/CPN [12] relied on the GAP programming enviroment [7] for efficient manipulation of algebraic groups. More generally, Comms/CPN makes it possible to integrate Design/CPN and external applications via TCP/IP.

Comms/CPN was developed as a Practical Industrial Experience project in Computer Systems Engineering at the University of South Australia. The development is part of a research project being undertaken by the Air Operations Division (AOD) within the Australian Defence Science and Technology Organisation (DSTO) [24] and the Computer Systems Engineering Centre (CSEC) [3] at the University of South Australia. It involves the modelling and analysis of Avionics Mission Systems (AMS) for testing and evaluation. Part of this research involves providing visualisation of the simulation of CPN models by extending them with application specific graphics. The external communication facilities provided by Comms/CPN allow this visualisation to take place using an external visualisation package. The external visualisation package itself is currently in the process of being developed, but a prototype demonstrating a proof-of-concept exists.

The development of Comms/CPN is based upon previous work, in particular the Master's thesis [19]. The work done in [19] however has some drawbacks as it was primarily an encapsulation of TCP/IP. Only one connection could be opened, and this connection is made to a location fixed at compile time, i.e. it could not be changed without re-switching the CPN model. Comms/CPN extends the work presented in [19] in several ways. Firstly, Comms/CPN allows dynamic creation of connections (also during the simulation of a CPN model), the external process to which connections are being made is not fixed, and multiple simultaneous connections are supported. Secondly, Comms/CPN implements a protocol on top of TCP/IP for passing messages between Design/CPN and the external application. It is planned to make Comms/CPN available for public use via the Design/CPN home page [21].

This paper is organised as follows. Section 2 provides a description of the design and requirements of the Comms/CPN library. Section 3 describes the implementation of the

COMMS/CPN library. An example of the use of COMMS/CPN for visualisation of the simulation of an AMS CPN model is presented in Section 4. Section 5 sums up the conclusions and outlines future work in further developing COMMS/CPN. The reader is assumed to be familiar with CPN models and the DESIGN/CPN tool.

2 Design Overview and Requirements

COMMS/CPN is designed to act as an interface between CPN models and TCP/IP. Figure 1 shows the overall architecture of COMMS/CPN and how it relates to DESIGN/CPN and TCP/IP. COMMS/CPN consists of three main modules, organised as layers. The *Communication Layer* contains the interface to the underlying transport protocol, in this case TCP/IP, and contains all TCP/IP and socket related primitive functions. The *Messaging Layer* is responsible for transforming the reliable byte stream service provided by the transport layer into a service suitable for passing *messages* between DESIGN/CPN and external applications. The *Connection Management Layer* allows users to open, close, send to, and receive from multiple connections. The Connection Management Layer is the layer that the CPN model will normally interface to. When relating this to the Open Systems Interconnectivity (OSI) model [26], COMMS/CPN can be viewed as the session layer. The Communication Layer provides an interface to TCP/IP, the transport layer of the OSI model. The Connection Management Layer provides an interface to DESIGN/CPN, the presentation layer.

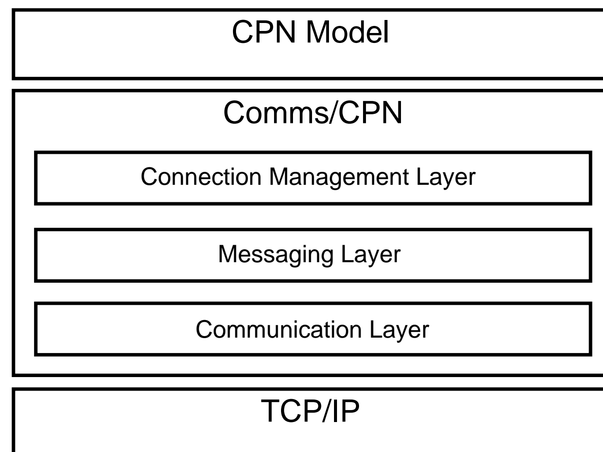

Fig. 1. Overall Architecture of COMMS/CPN.

The design of all three layers has been based on five functional requirements representing the services expected of the library. They are that COMMS/CPN shall provide means for CPN models to open connections to external processes, accept incoming connections requested by external processes, send data to external processes, receive data from external processes, and close connections to external processes. COMMS/CPN has been designed as an SML library to reflect the non-functional requirements of allowing easy integration with the DESIGN/CPN tool, and so that the communication facilities are separate from the DESIGN/CPN GUI. This allows the communication facilities to be accessed using library functions that can be included in code segments of transitions, auxiliary boxes, or in the top loop of the DESIGN/CPN simulator.

During the design and requirements processes of this architecture, three key design issues were identified. These are the *Fundamental Method of Communication*, *Connections and Connection Management*, and *Data Transfer*. The design and requirements reflect a desire to make Comms/CPN applicable to the widest range of applications possible. In the following subsections we discuss each of these design issues in detail.

2.1 Fundamental Method of Communication

The design decision was made for the underlying communication protocol to be TCP/IP. This comes from the requirement that data transmissions must be error free and in-order, and TCP/IP is a transport protocol that achieves this. TCP/IP is also desirable as it is a standard protocol, and most devices implement a TCP/IP stack and most programming environments (including SML) provide an interface to it through TCP/IP sockets.

Master thesis [19] examined the use of TCP/IP to communicate between Design/CPN and external processes. It summarised the ideas and concepts from various papers and publications. Of particular importance was [14], focussing on interaction between Design/CPN and Java processes. Two solutions were presented in [19] as to how TCP/IP communication between Design/CPN and external processes could be realised. The first was called the *Pure TCP Solution* in which Design/CPN connected directly to external applications via TCP/IP using functions in a communication library. The second was called the *Messenger Solution* in which library functions were used to communicate (via Unix pipes) with an external messenger subprocess written in Java. The Java subprocess then used TCP/IP to communicate with external processes.

Both the Pure TCP and Messenger solutions would provide usable communication facilities. The fundamental design choice for Comms/CPN was made to choose the Pure TCP solution for the design of Comms/CPN. Pure TCP provides for easier and tighter integration with Design/CPN, and can be used wherever Design/CPN is used. The Messenger solution (following the suggested implementation in [19]) does not provide for easy or tight integration because it uses a programming language other than SML. The Communication Layer reflects this decision. This layer provides an interface to TCP/IP for the Comms/CPN library. The primitives provided in this layer are used by the Connection Management Layer to establish TCP/IP connections to external processes, send and receive data in the form of streams of bytes, and to close TCP/IP connections to external processes. Moreover, it is possible to implement an architecture similar to the the Messenger solution of [19] based purely on Comms/CPN.

2.2 Connections and Connection Management

A connection represents a communication channel between a CPN model and an external process. It is the Connection Management Layer that manages connections between CPN models and external processes by creating, storing, and removing connection information. Non functional requirements of the library state that the library must be capable of handling multiple connections, and that these connections can be established dynamically during the simulation of a CPN model. Also, requirements state that the library must provide a mechanism for identifying connections and abstracting from low level socket identifiers. The design of the Connection Management Layer reflects these requirements.

Connection Identification. The connection identification strategy adopted in the design of the Connection Management Layer is to assign a unique string to each connection as it is made. String identifiers offer the advantage of being more human-readable and recognisable than an integer or a low level socket identifier. A string can be provided by the user (to further aid in recognisability) or it could be provided internally by COMMS/CPN. Strings can easily be used as tokens within a CPN model to pass connection identifiers around during simulation of the CPN model.

Connection Attributes. When a connection is created, it is necessary for information about this connection to be recorded. These *connection attributes* must allow the connection to be identified and used. In order to identify the connection, the unique string identifier must be stored, and in order to use the connection, the low level TCP/IP socket identifier must be stored. The unique string identifier allows connections to be identified within CPN models, and the low level TCP/IP socket is needed in order to send and receive data. Without recording these two pieces of information, the establishment of connections becomes useless as there is no way to identify them or to use them. Before a connection is established, a check is made to ensure that the given unique string identifier is in fact unique. If not, the connection is not established. Multiple connections can be open simultaneously, so a data structure is needed to store the connection attributes of more than one connection. The Connection Management Layer contains a mechanism to do this, called the *Connection Storage Mechanism*, and a data structure in which the attributes are stored. The data structure must allow new information to be stored, existing information to be retrieved, and old information to be removed.

2.3 Data Transfer

A non functional requirement of COMMS/CPN is that the library must have the capability to send and receive all types of data, including user defined types (colour sets). This is important in increasing the overall usefulness of COMMS/CPN. TCP/IP dictates that data must be in the form of a sequence of bytes for transmission across a network, so data objects must be converted into this form for transmission.

The Messaging Layer of COMMS/CPN within the Connection Management Layer provides a solution. Generic send and receive functions that send and receive sequences of bytes, regardless of the type of the data objects being transmitted, can be written and included in the Connection Management Layer. This provides a way for users to send and receive sequences of bytes without having the responsibility of implementing the actual sending and receiving functions themselves.

In order to convert data objects to and from sequences of bytes, encoding and decoding is necessary. An encoding function converts a data object into a form suitable for transmission via TCP/IP, and a decoding function converts a sequence of bytes into a data object. SML, being a functional programming language, allows functions to be passed as parameters. In this way, encoding and decoding functions can be written for any data type desired, and can then be passed as parameters to the generic send and receive functions. The generic send function applies the encoding function to a data object in order to convert it to a suitable form for transmission. Similarly, the generic receive function applies the decoding function to a received sequence of bytes to form data objects. In this way, any type of data can be sent or received, provided the corresponding encoding and decoding functions have been written. Encoding and decoding functions for commonly used data types are supplied with the library, e.g. for strings and integers.

Another non functional requirement is to ensure that data sent and received has a consistent format regardless of its type. The virtual byte stream service provided by TCP/IP allows for the transmission and reception of sequences of bytes. However, when dealing with many different types of data (including user defined types) this is not always adequate. A more structured approach is required to delineate items of data from the virtual byte stream to provide a better service than just a stream of bytes. Data needs to be packetised for transmission so that when an item of data is sent, the receiver knows when all of it has arrived. The solution to achieve this atomicity is to segment the stream of bytes, and to provide each segment with a header that describes it. A segment of bytes (payload) together with its header make up a data *packet.* A message is one or more of these packets.

The packet format chosen for COMMS/CPN consists of a one byte header and a maximum of 127 bytes of payload data. This is illustrated in Figure 2. Seven bits of the header indicate the length of the payload data attached to it (i.e. $2^7 - 1 = 127$ bytes) and the remaining bit indicates whether this is the last packet in the transmission of the data item, in the case where a data item is greater than 127 bits in length. In this way, the header allows variable length data packets to be handled. It must be stressed that the maximum of 127 bytes of payload data can easily be changed by choosing a different sized header. What is important is that the peer entity in the external process (with which communication is taking place) implements the same protocol at the Messaging Layer.

The choice of a one-byte header is somewhat arbitrary, as there does not appear to have been any studies conducted regarding ideal packet length when transferring data between DESIGN/CPN and external processes. It should be mentioned that other segmentation and assembly protocols could be used to achieve the same service.

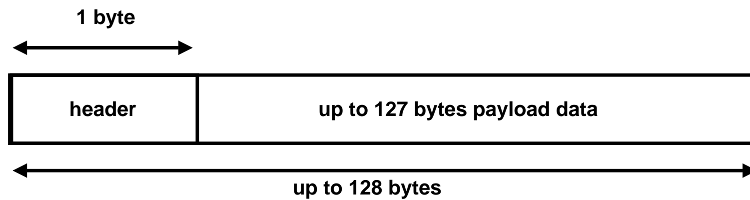

Fig. 2. Packet format for transmission of data.

3 Implementation

This section describes the implementation of the design into a working STANDARD ML [8] library. The implementation consists of SML library files, one for each of the layers described previously in Sect. 2.

3.1 The Communication Layer

The Communication Layer is implemented with TCP/IP as the underlying transport protocol. It is designed to encapsulate the TCP/IP protocol and to provide users of this layer with a shielded interface to the network functions provided by TCP/IP. The SML/NJ standard library [25] contains a structure called *Socket* in which primitive operations on sockets are available. The Communication Layer is based on this library.

Figure 3 lists the *COMMS_LAYER* SML signature. This signature is implemented by the *C*ommsLayer structure constituting the Communication Layer. We describe each of the primitives provided by the Communication Layer in more detail below.

```
signature COMMS_LAYER =
  sig
    type channel
    exception BadAddr of string

    val connect : string * int -> channel
    val accept : int -> channel

    val send : channel * Word8Vector.vector -> unit
    val receive : channel * int -> Word8Vector.vector

    val disconnect : channel -> unit
  end;
```

Fig. 3. SML signature for Communication Layer.

Most of the implementation of the Communication Layer comes directly from [25]. The only new datatype introduced is called *channel*. Its purpose is to allow the Connection Management Layer to map string identifiers to TCP/IP sockets without using any TCP/IP related code. Below we give a brief description of each of the primitives.

connect Creates a connection, acting as a client, to an external process. The first argument is used to specify the hostname of the external application, and the second argument is used to specify the port number.

accept Waits for an incoming connection on the port specified as the argument. The primitive blocks until an external application connects, and the connection is then established.

send Sends the sequence of bytes specified as the second argument on the channel specified as the first argument.

receive Receives the number of bytes specified as the second argument on the channel specified as the first argument. The primitive will block until the specified number of bytes have been received on the channel.

disconnect Closes the connection specified as the argument.

3.2 Messaging Layer

The Messaging Layer is implemented on top of the Communication Layer. Figure 4 lists the *MESSAGING_LAYER* SML signature. This signature is implemented by the *M*essagingLayer structure constituting the Messaging Layer. The *send* function implements the transmission of messages (specified as a sequence of bytes) according to the protocol discussed in Sect. 2.3. The data provided is segmented and appropriate headers are added to each segment. This forms packets of data that are sent to the external process using the *send* function from the Communication Layer. The *receive* function implements the reception of messages. It reads one byte (the header byte) and the corresponding number of payload bytes from the channel using the *receive* function from the Communication Layer. The *InvalidDataExn* exception will be raised if received data does have the format specified in Fig. 2.

```
signature MESSAGING_LAYER =
  sig
    type channel
    exception InvalidDataExn of string

    val send : channel * Word8Vector.vector -> unit
    val receive : channel -> Word8Vector.vector
  end
```

Fig. 4. SML signature for Messaging Layer.

3.3 Connection Management Layer

The Connection Management Layer builds on top of the Communication and Messaging Layers by providing the ability and interface to communicate with multiple external processes. The connection storage mechanism is implemented in this layer. The Connection Management Layer is implemented independently of TCP/IP and sockets. Instead it uses the services provided the Communication Layer and the Messaging Layer to interact indirectly with TCP/IP. It is the functions in this layer that would normally be used in a CPN model.

Figure 5 lists the *CONN_MANAGEMENT_LAYER* SML signature. This signature is implemented by the *C*onnManagementLayer structure constituting the Connection Management Layer. The signature specifies the type *Connnection* used to identity connections. The type has been implemented as strings. We describe each of the primitives provided by the Communication Layer in more detail below.

```
signature CONN_MANAGEMENT_LAYER =
  sig
    type Connection
    exception ElementMissingExn of string
    exception DupConnNameExn of string

    val openConnection : Connection * string * int -> unit
    val acceptConnection : Connection * int -> unit

    val send : Connection * 'a * ('a -> Word8Vector.vector) -> unit
    val receive : Connection * (Word8Vector.vector -> 'a) -> 'a

    val closeConnection : Connection -> unit
  end
```

Fig. 5. SML signature for Connection Management Layer.

openConnection Allows users to connect to external processes as a client. It takes three input parameters. The first of these is the unique string identifier (of type *Connection*) to be associated with the new connection. The second and third are the host name and port number that make up the address of an external process. The function first checks to ensure the string identifier given is unique, by searching the existing connections. A *DupConnNameExn* exception is raised if this is not the case. The function then attempts to create a connection to the external process by using the primitives from the Commu-

nication Layer. If successful, the appropriate information is stored and added to the list of connections. The return type of this function is type unit.

acceptConnection Provides server behaviour, and allows external processes to connect to Design/CPN. This function takes a *Connection* (string identifier) and a port number as input. The function checks that the given string identifier is unique, and then listens on the given port for incoming connection requests. This causes Design/CPN to *block* until an incoming connection request is received. When this happens, a connection is established with the external process requesting the connection.

send Allow users to send any type of data to external processes. The function is polymorphic, in the sense that the data passed to it for sending can be of any type, including user defined types. Three parameters are passed to this function as input. The first is a string identifier for the connection, the second is the data to send, and the third is a function to encode the data to send. The purpose of the encoding function is to encode the data to send into a sequence of bytes. This allows the data to be of any type, provided an encoding function exists for that type. The *send* function retrieves the connection corresponding to the given string identifier. It then invokes the *send* primitive at the Messaging Layer. The return type of this function is type unit.

receive Allows users to receive any type of data from an external process. The *receive* function is polymorphic in the same way as the *send* function. The parameters to this function are a string identifier and a decoding function, to decode the received byte vector into the appropriate data type. The function begins by retrieving the connection from which data will be received. It then invokes the *receive* from the Messaging Layer to receive the data. The payload data (which was stored in the correct order when it was read) is then passed to the decoding function. The resulting decoded data is then returned.

closeConnection Allows users to close a connection. The string identifier of the connection to be closed is passed to this function as the argument. A search of the connections is conducted to ensure that a connection exists with that string identifier. If the connection does not exist, an *ElementMissingExn* exception is raised. The connection itself is closed by calling the *disconnect* function from the Communication Layer. The stored connection information is then removed from the list of connections. The return type of this function is type unit.

4 Application of Comms/CPN

An external communication infrastructure was required as part of the research project on modelling and analysis of avionics mission systems mentioned in the introduction. Comms/CPN was developed for the purpose of providing external visualisation of the simulation of an Avionics Mission System (AMS) CPN model.

Figure 6 illustrates the architecture of the visualisation facilities, and how Comms/CPN fits into this architecture. The idea is that Comms/CPN will provide the necessary communication infrastructure to allow data to be sent from the CPN model to an external animation package which will then interpret this data and update the animation as necessary. The architecture consists of three applications (processes). The Design/CPN GUI (left), the SML process (middle), and the external Visualisation Package (right). The Design/CPN GUI and the simulator part of the SML process communicates (in the usual way) for visualising the token game during simulation in the Design/CPN GUI. This communication is done via TCP/IP using the DMO module of the Design/CPN simulator. In addition to this, the

simulator part of the SML process now also communicates with the external Visualisation Package via Comms/CPN.

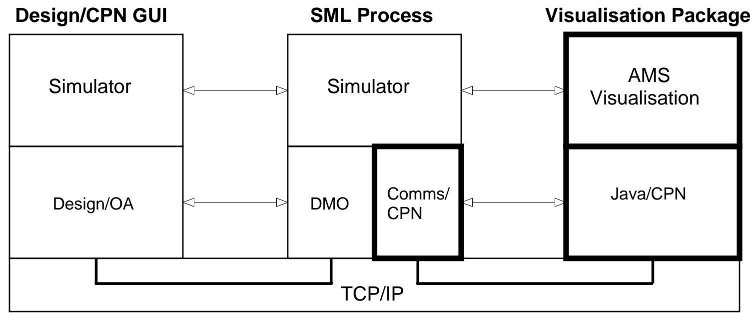

Fig. 6. Comms/CPN in the context of external visualisation.

The external visualisation package is currently under development. It is being implemented in Java [10] and consists of two main modules. The *AMS Visualisation* module is the module that provides the visualisation facilities. This module has been implemented using the Java Swing library [9]. The Java/CPN module is the peer module of Comms/CPN at the Java side. The Java/CPN module contains primitives similar to those in Comms/CPN to enable communication, and implements the protocol described in Sect. 2.3. We describe the Java/CPN module, the AMS visualisation module, and how it interacts with the AMS CPN model in more detail in the following subsections.

4.1 Java/CPN

The purpose of Java/CPN is to allow Java processes to communicate with Design/CPN through Comms/CPN. The current implementation of Java/CPN is the minimal implementation necessary to enable communication. It incorporates the equivalent functionality of the Messaging and Communication layers from Comms/CPN. The Communication Layer functionality from Comms/CPN and TCP/IP is already encapsulated in the *Socket* objects provided by Java through the use of Socket methods and the input and output streams available from the socket itself.

No connection management has been implemented within Java/CPN as this is a minimal implementation, however the important thing is that it implements the same protocol as the Messaging Layer from Comms/CPN as described in Sect. 2.3 . The interface of Java/CPN is shown in Figure 7. As in Comms/CPN, generic send and receive functions have been provided at the level of the Messaging Layer, meaning that sequences of bytes are passed to the *send* method and returned from the *receive* method. The *connect*, *accept*, and *disconnect* methods have been provided at the level of the Communication Layer from Comms/CPN. The deliberate attempt was made to make the interface as close to that of Comms/CPN as possible. We describe each of the methods within Java/CPN in more detail below.

The *connect* method acts in the same way as the *connect* method from the Communication Layer of Comms/CPN. It takes a host name and port number as arguments, and attempts to establish a connection as a client to the given port on the given host. This method does not return a value. Once the connection has been established (i.e. the socket opened) input

```
import java.util.*;
import java.net.*;
import java.io.*;

public interface JavaCPNInterface
{
  public void connect(String hostName, int port);

  public void accept(int port);

  public void send(ByteArrayInputStream sendBytes) throws SocketException;

  public ByteArrayOutputStream receive() throws SocketException;

  public void disconnect();
}
```

Fig. 7. Interface to Java/CPN.

and output streams are extracted from the socket to enable the transmission and reception of bytes.

The *accept* method also acts in the same way as the *accept* method from the Communication Layer of COMMS/CPN. It takes a port number as an argument and, acting as a server, listens on the given port number for an incoming connection request. When received, it establishes the connection. Again, once the connection has been established, input and output streams are extracted from the socket to enable the transmission and reception of bytes. This method does not return a value.

The *send* method takes a *ByteArrayInputStream* object (a Java object for holding sequences of bytes, acting as input) as the argument. The segmentation into packets occurs in a similar way to that which occurs in the Messaging Layer of COMMS/CPN. Bytes are read from the *ByteArrayInputStream* object, a maximum of 127 at a time, and a header added as described in Sect. 2.3. The data packets formed are then transmitted to the external process through methods acting on the output stream of the socket. The *send* method does not return a value.

The *receive* method has no arguments. It uses methods that act on the input stream of the socket to firstly receive a header byte, and then receive the number of payload bytes specified in the header, from the external process. The payload bytes are stored in a *ByteArrayOutputStream* object (a Java object for storing bytes as output) as each segment of payload data is received. This process is repeated until all data has been received for the current implementation. The receive method returns the *ByteArrayOutputStream* object.

The *disconnect* method has no arguments, and returns no value. It acts in the same way as the *disconnect* function from the Communication Layer of COMMS/CPN, except that it also closes the input and output streams from the socket before the socket itself is closed.

Methods external to the JAVA/CPN class must be used to convert from data (i.e. a string) into a *ByteArrayInputStream* object, and from a *ByteArrayOutputStream* object back into data. This is akin to the encoding and decoding functions passed into the send and receive functions of the Connection Management Layer in COMMS/CPN.

4.2 Visualisation of Avionics Mission Systems

An Avionics Mission System (AMS) consists of a number of subcomponents connected via a serial data bus. The serial data bus (SDB) is controlled by the Mission Control Computer (MCC), and subcomponents communicate by the exchange of data across the SDB. An initial CPN model of a generic AMS [16,22] has been constructed, capturing the AMS at a high level of abstraction, including communication between subcomponents. In this section we show how COMMS/CPN can be used to visualise this communication.

A snapshot from a prototype display of the visualisation package is shown in Figure 8. The display shows the various subcomponent of the AMS connected to the SDB. Each time two components communicate via the SDB, the external visualisation package will show this communication by highlighting the two subcomponents and the SDB. The simulation will then block until the user clicks on the *Continue* button.

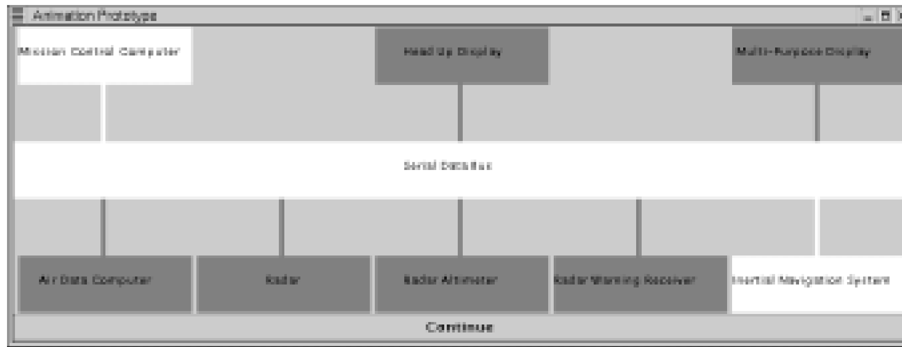

Fig. 8. Snapshot from the external visualisation package.

In order to provide external communication facilities, the COMMS/CPN library must be included in the CPN model. This consists of loading a number of SML files using the SML *use* command. For the AMS CPN model, the visualisation is done using code segments attached to the transitions. Opening and closing the connection to the external Visualisation Package is done by evaluating SML code in auxiliary boxes.

Of particular interest in providing visualisation of SDB communication is the *SerialDataBus* subpage of the AMS CPN model. This page is shown in Figure 9. Each subcomponent of the AMS has a unique address associated with it, and the *Transmit* transition on this subpage models the actual transmission of messages across the SDB. Two auxiliary boxes containing COMMS/CPN primitives have been added to the top left of this page. When evaluated, the first opens a connection using the *openConnection* primitive in the *ConnManagementLayer* structure, and the second closes the connection using the *closeConnection* primitive in the *ConnManagementLayer* structure which constitutes the Connection Management Layer of COMMS/CPN.

A code segment has been attached to the *Transmit* transition. This code segment calls the function shown in Figure 10. The purpose of this function is to take the addresses of the sender and receiver, map them to integers, transmit these two integers to the external visualisation package, and then await a response before continuing the simulation. The external visualisation package interprets the two integers as the corresponding sender and destination and updates the animation to reflect this data transfer. The COMMS/CPN *send* primitive is

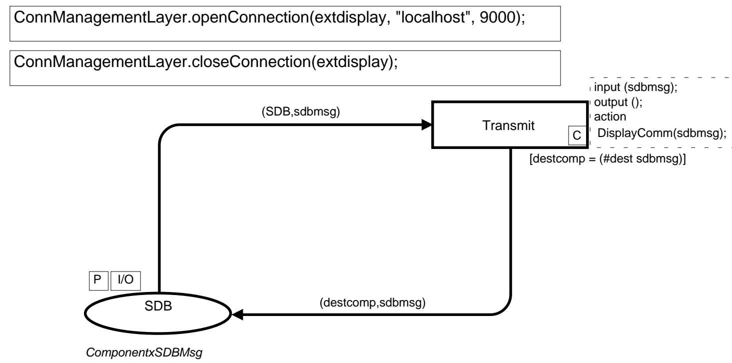

Fig. 9. The SerialDataBus page of the AMS CPN model.

used to send the two integers, and the *receive* primitive is used to receive the response from the visualisation package caused by the user clicking on the *Continue* button.

```
val extdisplay = "extdisplay"; (* --- name of connection --- *)

(* --- map subcomponent to an identifier --- *)
fun ComponentOpcode (MCC _) = "1"
  | ComponentOpcode (DISPLAYPROC (DISPLAY HUD)) = "2"
  | ComponentOpcode (DISPLAYPROC (DISPLAY MPD)) = "3"
  | ComponentOpcode (SENSOR ADC) = "4"
  | ComponentOpcode (SENSOR RADAR) = "5"
  | ComponentOpcode (SENSOR RALT) = "6"
  | ComponentOpcode (SENSOR WRW) = "7"
  | ComponentOpcode (SENSOR INS) = "8";

(* --- update the external display and wait for Continue --- *)
fun DisplayComm ({src,dest,...} : SDBMsg) =
    let
        val (src', dest') = (ComponentOpcode src, ComponentOpcode dest)
    in
        ConnManagementLayer.send(extdisplay,src'^","^dest',stringEncode);
        ConnManagementLayer.receive(extdisplay, stringDecode);
        ()
    end;
```

Fig. 10. SML functions for visualising SDB communication.

5 Conclusions and Future Work

Comms/CPN originated from a desire to provide visualisation of the simulation of AMS CPN models. Existing methods of visualisation were not satisfactory in this case, either through being too detailed or by having limited capability and being tied to the Design/CPN GUI. By developing an external visualisation package, access to greater graphical capabilities becomes

possible and visualisation is no longer tied to the Design/CPN GUI. Comms/CPN provides the necessary communication infrastructure to allow the external visualisation to take place.

The functional and non functional requirements of this library were considered and it was determined that five main functions must be provided, i.e. opening and accepting connections, sending data to and receiving data from external processes, and closing connections. It was also determined that the library must support multiple simultaneous connections and allow dynamic creation of connections. From these requirements, the library was designed. Three areas of design were considered. They were the fundamental method of communication, connection management, and data transfer. The architecture of Comms/CPN was defined to consist of three layers, sitting between Design/CPN and TCP/IP. The Communication Layer provides the interface to TCP/IP, the Messaging Layer introduces message passing scheme, and the Connection Management Layer provides the interface to Design/CPN.

The current implementation of the library poses some difficulties when it comes to accepting incoming connection requests and receiving data. When listening for an incoming connection request, Design/CPN blocks causing the simulation of the CPN model to block also. The same situation occurs when a receive operation is called but there is no data to receive. This blocking property is unfortunate if Design/CPN is performing a simulation, because it causes the entire simulation to block (as Design/CPN is purely single threaded.)

One possible area for investigation in the future is to provide non-blocking options for both the receive and accept operations. This is one area where using a messenger subprocess would have provided a relatively simple solution, as discussed Sect. 2. There is a possibility that in the future, Comms/CPN could be used in conjunction with an external subprocess, to form a hybrid Pure TCP and Messenger solution. A system call similar to the *select* call in the C programming language would also provide a solution. Using Concurrent ML (CML) [5] instead of SML as the programming language for the Design/CPN simulator and for Comms/CPN would eliminate the blocking issues and so would also provide a solution.

Currently, the library only facilitates each connection to be connected to a single external process. To connect to more than one external process, multiple connections are used. It may be possible to extend the library functions to allow *multicasting* whereby more than one external process can receive the same data from a single connection. In this way multiple external processes will receive exactly the same data. Such multicasting would be useful when using this library for the purposes of visualisation of Design/CPN simulations because it would allow exactly the same visualisation to be seen on different remote machines.

When a connection is created, the current implementation requires the user to provide the unique identifier. Future implementations of this library may give Design/CPN the ability to provide this unique identifier itself, and to return this automatically generated identifier to the user for subsequent use.

Another area of future development would involve the creation of communication modules like Java/CPN for other programming languages, such as C/CPN, Perl/CPN and so on. Another issue to consider as part of future work is to make Comms/CPN and Java/CPN libraries more fault tolerant.

Acknowledgments. The work presented in this paper was supported by the Australian Defence Science and Technology Organisation (DSTO) under contract no. 687237, and by a Divisional Small Grant from the University of South Australia. The authors also acknowledge valuable comments and feedback from Prof. Jonathan Billington.

References

1. Animation by Mimic/CPN. http://www.daimi.au.dk/designCPN/libs/mimic/.
2. C. Capellmann, S. Christensen, and U. Herzog. Visualising the Behaviour of Intelligent Networks. In *Services and Visualisation, Towards User-Friendly Design*, volume 1385 of *Lecture Notes in Computer Science*, pages 174–189. Springer-Verlag, 1998.
3. Computer Systems Engineering Centre. http://www.unisa.edu.au/eie/csec.
4. D. E. Comer. *Computer Networks and Internets*. Prentice-Hall International, Inc., 1997.
5. Concurrent ml. http://cm.bell-labs.com/cm/cs/who/jhr/sml/cml/index.html.
6. G. Gallasch and L. M. Kristensen. Comms/CPN library. http://www.daimi.au.dk/designCPN/libs/commscpn/.
7. The GAP Group, Aachen, St Andrews. *GAP – Groups, Algorithms, and Programming, Version 4.2*, 1999. (http://www-gap.dcs.st-and.ac.uk/~gap).
8. R. Harper. *Programming in Standard ML*. School of Computer Science, Carnegie Mellon University, http://www.cs.cmu.edu/ rwh/introsml/, 2000.
9. Java swing library. http://java.sun.com/products/jfc/tsc/index.html.
10. java.sun.com - The Source for Java(TM) Technology. http://www.java.sun.com/.
11. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts*. Monographs in Theoretical Computer Science. Springer-Verlag, 1992.
12. J. B. Jørgensen and L. M. Kristensen. *Design/CPN Condensed State Space Tool Manual*. Department of Computer Science, University of Aarhus, Denmark, 1996. Online: http://www.daimi.au.dk/designCPN/.
13. L. M. Kristensen, S. Christensen, and K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2):98–132, 1998.
14. O. Kummer, D. Moldt, and F. Wienberg. A Framework for Interacting Design/CPN- and Java-Processes. In J. Kleijn and S. Donateli, editors, *Applications and Theory of Petri Nets*, volume 1639 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.
15. B. Lindstrøm. Web Based Interfaces for Simulation of Coloured Petri Net Models. In K. Jensen, editor, *Proceedings of Workshop on Practical Use of High-level Petri Nets*, pages 15–32. Department of Computer Science, University of Aarhus, Denmark, 2000. DAIMI PB-547. Available via http://www.daimi.au.dk/pn2000/proceedings/.
16. C. Douglass Locke, L. Lucas, and J. B. Goodenough. Generic Avionics Software Specification. Technical Report CMU/SEI-90-TR-8, Software Engineering Institute, Carnegie Mellon University, December 1990.
17. L. Lorentsen and L. M. Kristensen. Exploiting Stabilizers and Parallelism in State Space Generation with the Symmetry Method. In *Proceedings of International Conference on Application of Concurrency in System Design (ICACSD'2001)*, pages 211–220. IEEE Computer Society, 2001.
18. Design/CPN Message Sequence Charts library. http://www.daimi.au.dk/designCPN/libs/mscharts/.
19. S. Nimsgern and F. Vernet. Communication between Coloured Petri Net Simulations and External Processes. Master's thesis, Department of Computer Science, University of Aarhus, 2000.
20. Standard ML of New Jersey. http://cm.bell-labs.com/cm/cs/what/smlnj/.
21. Design/CPN Online. http://www.daimi.au.dk/designCPN/.
22. Z. Qureshi, L. M. Kristensen, and J. Billington. Towards Modelling and Analysis of Avionics Mission Systems using Coloured Petri Nets and Design/CPN. Technical report, Defense Science and Technology Organisation, 2001. Divisional Dissussion Paper.
23. J. L. Rasmussen and M. Singh. Designing a Security System by Means of Coloured Petri Nets. In *Proceedings of ICATPN'96*, volume 1091 of *Lecture Notes in Computer Science*, pages 400–419. Springer-Verlag, 1996.
24. Australien Defence Science and Technology Organisation. http://www.dsto.defence.gov.au.
25. SML/NJ library. http://cm.bell-labs.com/cm/cs/what/smlnj/doc/smlnj-lib/index.html.
26. W. Stallings. *Data and Computer Communications*. Prentice-Hall, 2000.
27. J. D. Ullman. *Elements of ML Programming*. Prentice-Hall, 1998.