Using Design/CPN to Design a Visualisation Extension for Design/CPN

Mathew Elliot, Jonathan Billington, and Lars M. Kristensen

Computer Systems Engineering Centre
School of Electrical and Information Engineering
University of South Australia
Mawson Lakes Campus, SA 5095, AUSTRALIA
Email: {Mathew.Elliot@dsto.defence.gov.au, Jonathan.Billington@unisa.edu.au, lars.kristensen@unisa.edu.au}

Abstract. This paper reports on the use of Coloured Petri Nets (CPNs) and DESIGN/CPN in the development of a protocol to support the visualisation and animation of a range of systems that can be modelled using CPNs. The developed protocol facilitates the co-ordination between a DESIGN/CPN simulation, and an *External Visualisation Package*. The protocol is modelled and analysed using CPNs created using DESIGN/CPN. The analysis shows that the protocol works as required and contains no deadlocks. The paper also analyses a more robust version of the protocol.

Keywords: Design/CPN Extensions, Visualisation, Animation of CPN Models

1 Introduction

When using the Coloured Petri Net (CPN) [7,8] tool, DESIGN/CPN [11], users are limited in the visualisations that can be performed. Whilst DESIGN/CPN provides very detailed visualisations in the form of the "token game" these are not easily understood by people unfamiliar with CPNs. There are other software packages that provide higher level visualisations for DESIGN/CPN, such as MIMIC/CPN [1] and the Message Sequence Chart library [10], however these packages are not always satisfactory for various reasons, including the fact that they rely on the DESIGN/CPN graphical user interface (GUI).

Prompted by the limitations imposed by the visualisation tools currently available for DESIGN/CPN we decided to create an *External Visualisation Package*. The key ideas behind this *External Visualisation Package* are: that visualisations can be developed with greater graphical capabilities (for example using Java libraries) than those of DESIGN/CPN; and the visualisation could be performed at a remote location, independently of the DESIGN/CPN GUI.

Such a visualisation tool has been created as part of the *Animation Facilities for Defence Systems Modelling* project [4] conducted by the University of South Australia for the Australian Defence Science and Technology Organisation (DSTO). The work was undertaken by a group of two students for their final year honours project, a mandatory component of a 4 year Bachelor of Engineering in Computer Systems Engineering at the University of South Australia. The project had two supervisors from the University of South Australia, and an external supervisor from DSTO, who provided requirements for the visualisation package. A requirement of final year computer systems engineering projects is that they must follow a full systems engineering approach. This involves fortnightly meetings with the supervisors, chaired by the students, to discuss project progress. Formal minutes of the meetings are taken

to record decisions and action items, and progress reports are submitted for each meeting. The students must submit for assessment a full set of documentation for the project that includes: a project plan, requirements specification, design description, test plan, test specification, test report, user manual, and system manual. The students also give a seminar on the project, and provide demonstrations of a working product, which is assessed by independent academic staff. In addition, honours students need to write a research proposal and paper.

The project officially comprises a quarter of a year's work for the two students, so that the total effort for this project was of the order of 5 person months. During this time, the students produced about 900 pages of A4 documentation for the project. This paper is a significant revision of the research paper written by the first author to fulfill the research paper requirement for honours students.

The goal of the project was to add visualisation facilities to a CPN model of an avionics mission system (AMS). An AMS is essentially a local area computer network consisting of a Serial Data Bus (SDB) that connects a set of components including a Mission Control Computer (MCC), various sensors, displays, and navigation systems.

The visualisation tool utilises the COMMS/CPN library [5, 6], to send information to the *External Visualisation Package (EVP)* via TCP/IP [2]. Further, an application protocol is required to support the exchange of visualisation commands between DESIGN/CPN and the EVP. The students developed and implemented a protocol for this purpose which is specified in the Software Design Description [3]. The students followed a rapid prototyping paradigm, to provide a rough first cut implementation to test out visualisation ideas, so that feedback could be obtained from the customer (DSTO) at the earliest opportunity. To add rigour to the approach for the honours component, it was decided to create a formal model of this application protocol using CPNs.

The purpose of this paper is to present our work on modelling this *visualisation* protocol and to verify that it is deadlock and livelock free and conforms to a sequencing constraint. The visualisation protocol is then extended and re-analysed.

The paper is organised as follows. Section 2 provides some insight into the nature of an AMS, and the visualisations that were implemented. We describe the visualisation protocol and its sequencing constraints in section 3 and a CPN model of the visualisation protocol in section 4. The model is analysed in section 5. Section 6 describes an enhanced protocol, which allows DESIGN/CPN to be informed of the success or failure of each visualisation command. The enhanced CPN model and its analysis are presented in sections 7 and 8, with our conclusions provided in section 9. The reader is assumed to be familiar with CPNs [7] and DESIGN/CPN [11].

2 Overview of the External Visualisation Package

The purpose of the external visualisation package is to provide domain specific visualisation facilities for the avionics mission system model at a higher, more abstract level than the DESIGN/CPN tool can provide. These facilities are to produce visualisations that are more strongly related to the physical system being modelled than is capable with DESIGN/CPN. This is important from a system development perspective as well as a marketing and management perspective because it is likely than these people would be unfamiliar CPN specific concepts. The four visualisations required to be supplied by the *EVP* were:

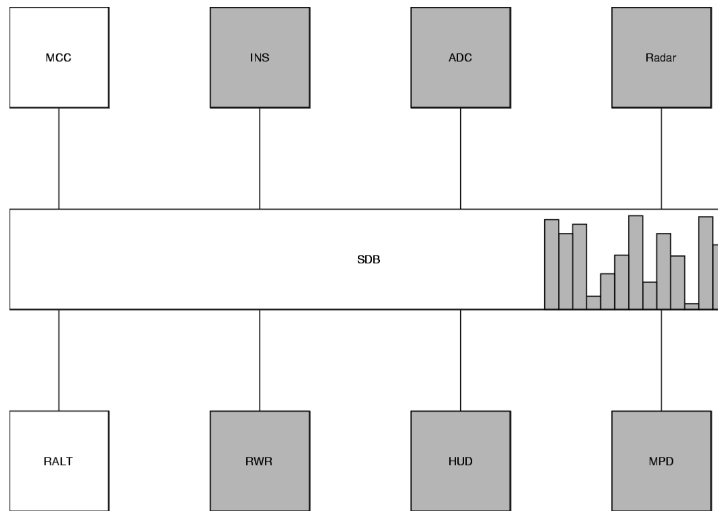

Fig. 1. Block diagram showing components of the AMS.

1. A block diagram of the AMS and its components. An example of this can be seen in Figure 1, which shows the Mission Control Computer (MCC), Inertial Navigation System (INS), Air Data Computer (ADC), Radar, Radar Altimeter (RALT), Radar Warning Receiver (RWR), Heads Up Display (HUD), Multi-Purpose Display (MPD), and Serial Data Bus (SDB) components of the AMS.
2. A diagram showing the progress throughout a mission. This is simply a bar chart with a line showing the progress through the mission as a percentage between 0 and 100.
3. More detailed information on the MCC and SDB components within the AMS as illustrated in Figure 2.
4. Message sequence charts to show instructions sent between components of the AMS.

A further decision was made to design the *EVP* architecture to be flexible enough to allow the use of different visualisation libraries. The rationale behind this was that the *EVP* could be reused to visualise many different systems rather than just avionics mission systems.

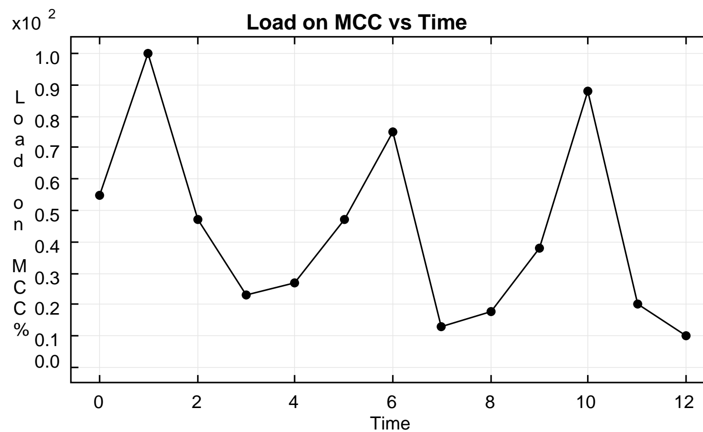

Fig. 2. Detailed view of the MCC component.

3 Visualisation Protocol

Figure 3 shows the overall architecture of the system that includes the *EVP*. The visualisation protocol is between the *AMS Visualisation Primitives* block on the DESIGN/CPN side and the *Visualisation Display* on the EVP side. The protocol itself comprises:

1. Client side: this is DESIGN/CPN in this implementation but could be any application.
2. Server side: the *EVP* which receives the visualisation instructions.

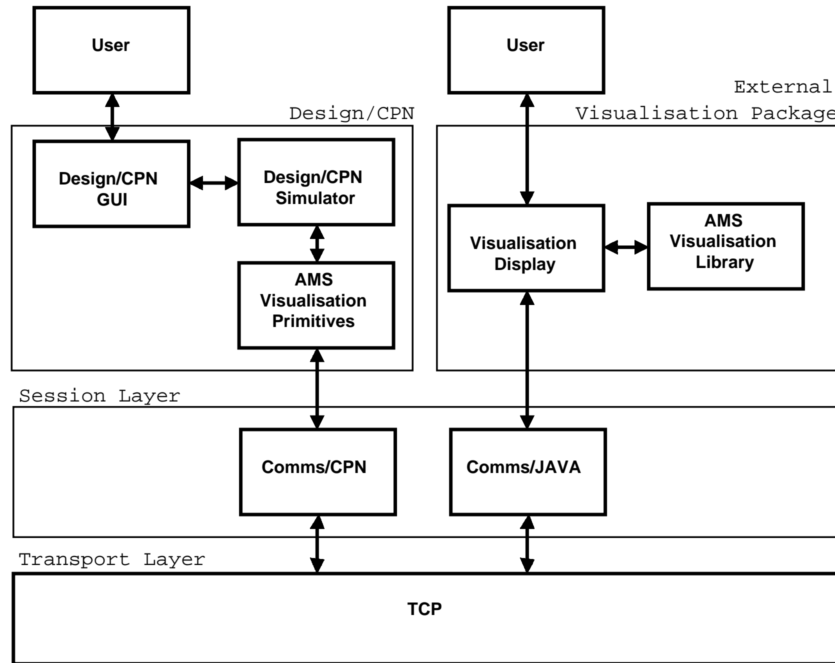

Fig. 3. Overall Decomposition of the External Visualisation Package.

3.1 Visualisation Instructions

A detailed specification for this protocol as used in the External Visualisation Package can be found in the Software Design Description [3]. The protocol facilitates the sending of instructions between the client, DESIGN/CPN, and the server, the External Visualisation Package. These instructions inform the visualisation package of the components required for visualisation and the visualisations that are to be performed. On the DESIGN/CPN side, the instructions are sent and received using the COMMS/CPN library [5,6] and hence communication between the client and server is via TCP/IP [2]. As TCP/IP is considered to provide a reliable transport service, it is assumed that all instructions sent arrive successfully and in order.

Two sets of instructions are sent between DESIGN/CPN and the External Visualisation Package. *Initialisation Instructions* as listed in Table 1 are used to initialise the External Visualisation Package before any visualisation can occur. The second set of instructions enables the *EVP* to perform animations involving the components specified using the initialisation commands. As the visualisation protocol is designed for visualisation of Avionics Mission Systems, many of these instructions are specific to these systems. The *Visualisation Instructions* are given in Table 2.

Instruction	Purpose
useLibrary	Specifies the visualisation library to be used by the *EVP*.
createComponent	Creates an instance of a component to be used by the *EVP*.
initComplete	Specifies that the initialisation is complete, (i.e. that all components to be used by the *EVP* have been created.)

Table 1. The *Initialisation Instructions* used within the visualisation protocol.

Instruction	Purpose
isCommunicating	displays that a specified component is communicating.
stoppedCommunicating	displays that a specified component is not communicating.
createLink	creates a visible link between two components used within a visualisation.
MCCLoadIndication	displays the load on the MCC component.
SDBLoadIndication	displays the load on an SDB component.
getInstruction	This is the only visualisation command that expects a response from the visualisation server. This instruction expects the server to reply with either a "STOP" or a "CONT" message. If a "STOP" signal is received the transfer of further visualisation commands is stopped. The "CONT" signal allows more visualisation commands to be sent.

Table 2. The *Visualisation Instructions* used within the visualisation protocol.

The order in which these instructions must be sent for successful visualisation is defined by the regular expression [9]:

$$useLibrary\ createComponent^{*}\ initComplete\ [VisualisationInstructions]^{*} \tag{1}$$

where [Visualisation Instructions] are any of the *Visualisation Instructions* defined in Table 2.

3.2 Client

The client Design/CPN sends out *Initialisation Instructions*, as defined in Table 1, to initialise the visualisation tool. It then sends *Visualisation Instructions* (listed in Table 2), until it is informed to stop by a reply of STOP (to the *getInstruction*) or there are no more visualisation commands to send. This progression of instructions is specified by the regular expression (Equation 1) given in section 3.1.

3.3 Server

The *Visualisation Server* receives instructions from Design/CPN via the session service. The server is responsible for determining if the instruction is valid (ie., is to be acted upon). If an instruction is determined to be invalid, (for example a useLibrary instruction is received after the library to be used has already been specified), the instruction is simply discarded.

4 Protocol CPN Model

4.1 Modelling Approach

Although the CPN model of the protocol is constructed on one level, (there are no substitution transitions), it is clearly separated into three sections: the Design/CPN Simulator; the

Session Service; and the External Visualisation Package, as can be seen in Figure 4. Figure 4 will be explained in more detail in the following subsections. We model the protocol at a systems design level and make the following assumptions:

1. The content of the instructions sent to and from the server are not important for the operation of the protocol.
2. Only one instruction can be processed at a time at the server.
3. The session service is free of errors, and preserves the sequence of messages.
4. A session is already established between both ends.
5. All the *Visualisation Instructions* defined in Table 2, except getInstruction as this requires a reply from the server, are abstracted into one message called UPDATEGUI. This is because these Visualisation Instructions just update the visualisation without waiting for a reply from the server. Therefore the behaviour for all of these instructions is the same.

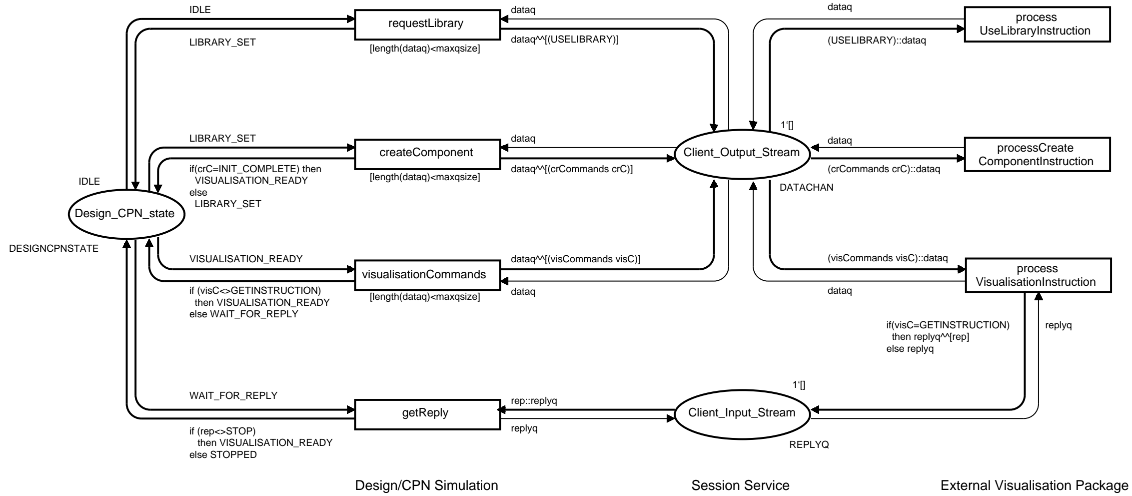

Fig. 4. Model of the basic protocol.

4.2 Data Structures and Declarations

Figure 5 shows the colour sets used in the CPN model. The CR_COMPONENT_COMMAND colour set is used to define the two initialisation instructions *createComponent and initComplete* as previously defined. The VISCOMMANDS colour set defines the visualisation instructions that can be sent to the *EVP*. The colour set DATA is the union of the initialisation instructions, the visualisation instructions and the *useLibrary* initialisation instruction, so that all types of instructions may be put in the client output queue. The DATACHAN colour set is simply a list of the DATA colour set, which is used to model the session service. The REPLY colour set enumerates the valid replies from the *EVP* (STOP or CONTINUE). The REPLYQ colour set is used to create a list of the REPLY colour set for the output queue of the *EVP*. The final colour set, DESIGNCPNSTATE, is used to enumerate the states of the Design/CPN side. The client is either IDLE, waiting to send the USELIBRARY initialisation instruction, LIBRARY_SET when the USELIBRARY instruction has been sent and the client is now either creating components or sending the *initComplete* instruction, VISUALISATION_READY, when sending visualisation instructions, WAIT_FOR_REPLY when

```
color CR_COMPONENT_COMMAND = with CREATECOMPONENT | INIT_COMPLETE;
color VISCOMMANDS = with GETINSTRUCTION | UD
color DATA = union crCommands : CR_COMPONENT_COMMAND + visCommands : VISCOMMANDS + USELIBRARY;
color DATACHAN = list DATA;
color REPLY = with CONTINUE | STOP;
color REPLYQ=list REPLY;
color DESIGNCPNSTATE = with IDLE | LIBRARY_SET | VISUALISATION_READY
                          | WAIT_FOR_REPLY | STOPPED;

val maxqsize=1;
var replyq : REPLYQ;
var rep : REPLY;
var dataq : DATACHAN;
var visC : VISCOMMANDS;
var crC : CR_COMPONENT_COMMAND;
```

Fig. 5. Colour set declarations.

waiting for a reply, or STOPPED when no more instructions are to be sent. A set of variables needed in arc expressions and guards, are declared, and the value of the constant, maxqsize, is defined, to limit the size of the session service queues.

4.3 Places

The DESIGN/CPN side of the protocol consists of a single place, *Design_CPN_state* that describes the state of the client. Its initial marking is IDLE. Two places (*Client_Input_Stream* and *Client_Output_Stream*) are used to represent the transmission of instructions between DESIGN/CPN and the *External Visualisation Package*. These two places have been implemented as FIFO queues (Assumption 3 in Section 4.1). The initial marking for both of these places is an empty list. The External Visualisation Package side contains no places and thus, no state information. This is due to the design of the server, which handles instructions received in any order.

4.4 Transitions

The DESIGN/CPN simulator side consists of four transitions. Three of these transitions are used to send instructions to the *External Visualisation Package* ordered to satisfy the regular expression shown in equation (1) in section 3.1. The *requestLibrary* transition is used to place a USELIBRARY instruction into the queue. The transition, *createComponent*, is used to put either a CREATECOMPONENT or INITCOMPLETE instruction into the queue. The *visualisationCommands* transition places either a GETINSTRUCTION or UPDATEGUI instruction into the queue. All three transitions have a guard to limit the number of instructions placed into the queue to prevent an infinite state space when performing analysis. The final transition, *getReply*, is used to receive a response from the EVP output stream.

The EVP side of the CPN contains three transitions *processUseLibraryInstruction*, *processCreateComponentInstruction*, and *processVisualisationInstruction*, which are used to handle the incoming commands. The *processUseLibraryInstruction*, and *processCreateComponentInstruction*, transitions simply remove the incoming instruction from the queue, as no information is sent back. The *processVisualisationInstruction* transition performs the same function;

however if a GETINSTRUCTION is received, either a STOP or CONT signal is sent back. This represents the user specifying whether to continue with the visualisation or stop, and is handled non determinisitically at this this level of abstraction.

5 Analysis of the Visualisation Protocol

Figure 6 shows the reachability graph of the CPN model described in section 3.1 using a queue size of one. As can be seen there is only one terminal marking for the CPN, which is located at node 11. The marking at node 11 shows that all the queues are empty and that only a STOPPED token exists in the *Design_CPN_state place*. This is the expected and desired result. The visualisation has ended with no instructions in the queues. This indicates that both ends are correctly synchronised. Node 11 is a home state, which is also desirable because this means that the protocol always has the possibility to terminate successfully from any state that the protocol may be in.

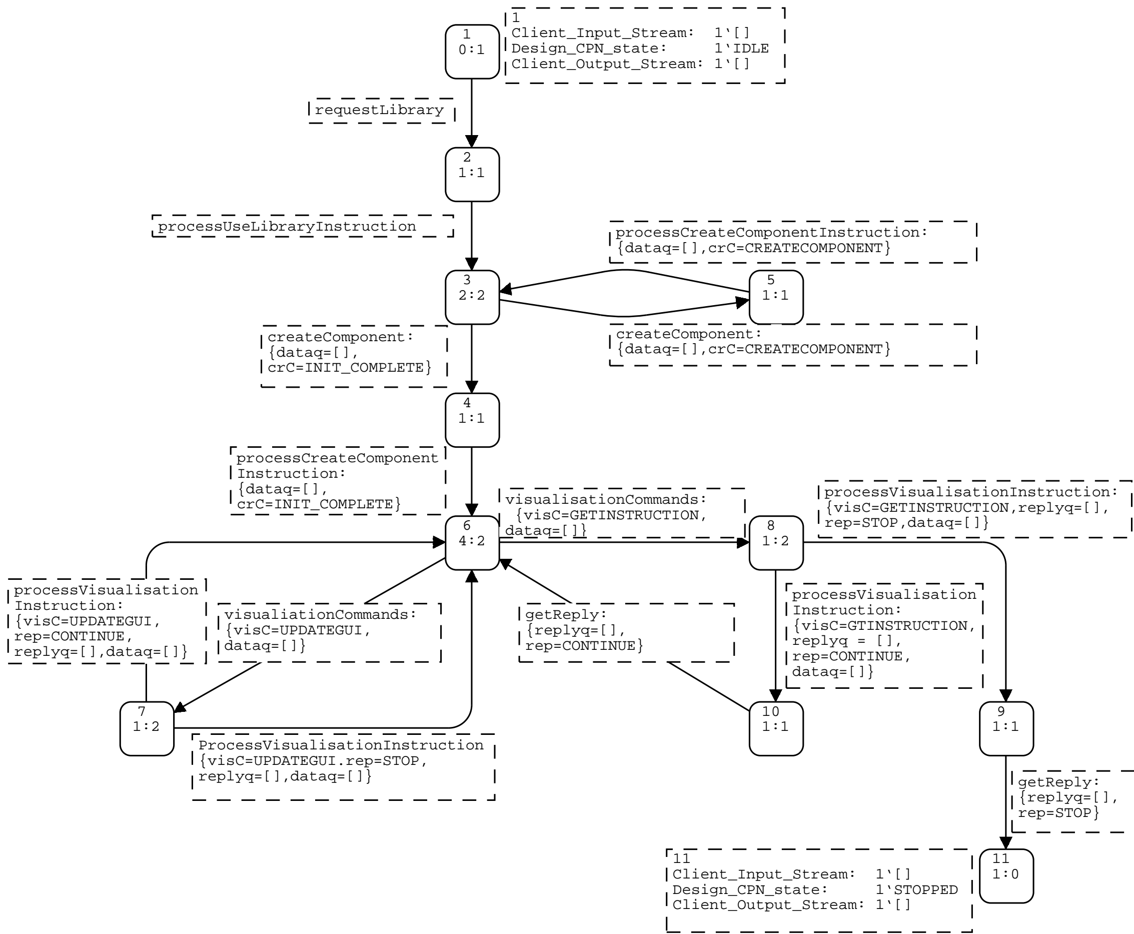

Fig. 6. Reachability graph of the Protocol CPN.

The *Strongly Connected Component (SCC)* graph was also calculated and is shown in Figure 7. The fact that the SCC graph contains less nodes than the reachability graph indicates that there are cycles within the protocol, although due to the fact that the terminal state

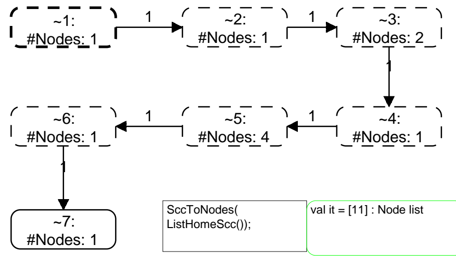

Fig. 7. SCC graph for the Basic Visualisation Protocol.

is a home state there always exists the possibility for the protocol to terminate successfully. An example of one of these cycles can be seen from nodes 6, 8 and 10 of the reachability graph. If the DESIGN/CPN simulator sends a GETINSTRUCTION and moves to the WAITING_FOR_REPLY state, then the *EVP* replies with CONTINUE, and the DESIGN/CPN simulator can again send the GETINSTRUCTION command. This cycle could continue forever. The protocol is proven to be livelock free because the SCC graph contains only one terminal node which contains the desired node 11 of the reachability graph.

Increasing the queue length beyond one causes the state space to grow. However in all examined cases up to a queue length of 40 (a node count of 3287) there exists only one terminal node with identical charateristics to node 11. We therefore conjecture that the queue size does not effect the operation of the protocol, but only affects the number of instructions that can be queued in the network at any time.

This model shows that the visualisation protocol satisfies its purpose of providing visualisation information to the *EVP* without being prone to deadlocks or livelocks, even though cycles are possible. However there is a major limitation of the protocol. This limitation is that the instructions sent are presumed to be correct. This means that users of the protocol are "left blind" not knowing if the instructions they just sent were valid. To overcome this, an attempt has been made to specify a more robust visualisation protocol, in Section 6, which sends replies back for each visualisation instruction received.

6 A More Robust Protocol for External Visualisation

The improved visualisation protocol includes a response for each instruction sent. This response indicates whether the instruction was successful or not, and if an instruction failed, the reason why is given to inform users of the protocol. The success or failure of these instructions is determined by the allowable sequences as specified by the regular expression in equation (1) of Section 3.1.

The *External Visualisation Package* side of the protocol has been modified significantly to determine if a visualisation instruction is acceptable. The server will either respond with S(uccess) or F(ail) depending on the current state of the *EVP* and the instruction received. The acceptability of an instruction is determined using the order that instructions can be received in for a successful visualisation to occur. This order is the same as the order specified in equation 1 using a FIFO queue ensures that instructions are received in the same order that they are sent.

The client has been modified to wait for a reply from the EVP each time an instruction is sent. The reply is then analysed to see if the instruction succeeded or failed, which allows users of the protocol to handle these events appropriately. This has the advantage of providing a flow

control, (which also eliminates the state space explosion), but also introduces round trip delay which could be significant on networks with large propagation delays. As the visualisation tool is intended to be used on a local area network this is not seen as a problem.

7 CPN Model of the Improved Protocol

Due to its greater complexity, the improved protocol CPN model has been constructed hierarchically. Figure 8 shows the top level of the hierarchy. The model of the improved visualisation protocol consists of four port places and four substitution transitions. The same assumptions have been made as were specified in Section 3.1.

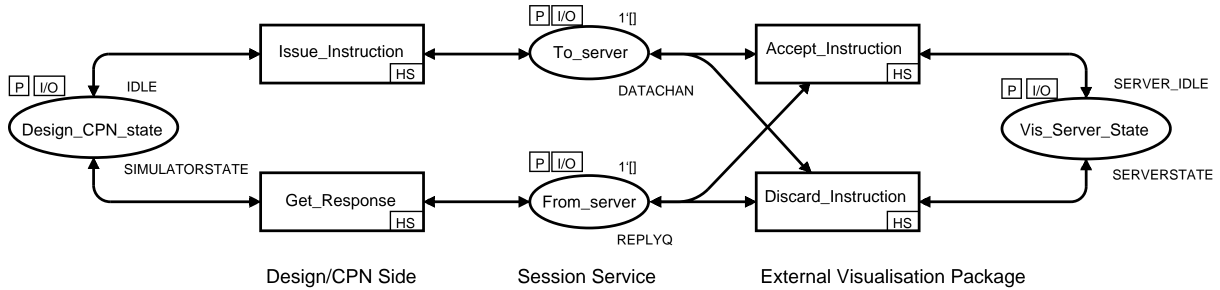

Fig. 8. Top Level of Improved Visualisation Protocol.

7.1 Data Structures and Declarations

Figure 9 shows the revised declarations. The changes with respect to Figure 5 are as follows.

The INSTRUCTION_STATUS colour set is used as a boolean to declare if an instruction was successful or not: **S** indicates success and **F** indicates a Fail. The colour set ARGS specifies the reason why an instruction failed. For example when a visualisation command is received before the initialisation is complete, the reason for failure would be INIT_NOT_COMPLETE. When an instruction is successful the ARG is NA (Not Applicable), unless it is a response to the AWAITING instruction in which case the ARG will be either CONT or STOP. The colour set REPLY is the product of the colour sets INSTRUCTION_STATUS and ARGS and defines a reply by the server. The SERVERSTATE colour set enumerates the three states of the Visualisation Package. SERVER_IDLE indicates when the server has not received a USELIBRARY instruction, LIBRARYSET indicates a USELIBRARY instruction has been received but the INITCOMPLETE instruction has not, and INITCOMPLETE indicates that the initialisation is complete.

The colour set SIMULATORSTATE is used to represent the various states of the Design/CPN side. This state is IDLE indicating that a USELIBRARY command has not yet been successfully sent, WAIT_FOR_LIB_REPLY when the client is waiting for a reply to a USELIBRARY instruction, LIBRARY_SET indicating that the USELIBRARY command has been successfully sent and the client will send either CREATECOMPONENT or INITCOMPLETE instructions, WAIT_FOR_CR_COMPONENT_REPLY indicating that the client is waiting for a reply to the CREATECOMPONENT, WAIT_FOR_INIT_COMPLETE_REPLY indicating that the client is waiting for a reply to an INIT_COMPLETE instruction, VISUALISATION_READY when visualisation instructions are being sent, WAIT_FOR_REPLY when the client is waiting for a response to a visualisation instruction and finally GET_RESPONSE when the client is waiting for a response to either continue or stop the visualisation.

```
color CR_COMPONENT_COMMAND = with CREATECOMPONENT | INIT_COMPLETE;
color VISCOMMANDS = with AWAITING | UPDATEGUI;

color DATA = union crCommands  : CR_COMPONENT_COMMAND +
                   visCommands : VISCOMMANDS + USELIBRARY;
color DATACHAN = list DATA;
color INSTRUCTION_STATUS=with S|F;

color ARGS= with LIBRARY_NOT_SET | LIBRARY_ALREADY_SET |
                 INIT_NOT_COMPLETE | NA | CONT | STOP;

color REPLY = product INSTRUCTION_STATUS*ARGS;
color REPLYQ=list REPLY;

color SERVERSTATE = with SERVER_IDLE | LIBRARYSET | INITCOMPLETE;
color SIMULATORSTATE = with  IDLE | LIBRARY_SET |
          VISUALISATION_READY | WAIT_FOR_REPLY | STOPPED | GET_RESPONSE |
          WAIT_FOR_CR_COMPONENT_REPLY | WAIT_FOR_INIT_COMPLETE_REPLY |
          WAIT_FOR_LIB_REPLY;

var crC : CR_COMPONENT_COMMAND;
var dataq : DATACHAN;
var visC : VISCOMMANDS;
var replyq : REPLYQ;
var rep : REPLY;
var sf : INSTRUCTION_STATUS;
var args : ARGS;
var serState: SERVERSTATE;

val maxqsize=1;
```

Fig. 9. Declarations for the CPN of Improved Visualisation Protocol.

7.2 Places

The DESIGN/CPN side has a single place, *Design_CPN_state*, used to record the current state of the DESIGN/CPN side of the protocol, according to its type, SIMULATORSTATE. The *To_server* and *From_Server* places shown in Figure 8 represent the *Client_Output_Stream* and *Client_Input_Stream* places as respectively discussed in Section 4. The External Visualisation Package has a single place to store the state of the *EVP* server.

7.3 Substitution Transitions

The Issue_Instruction subpage is shown in Figure 10. Its purpose is to place commands into the queue in the order specified in Section 3.1.

Figure 11 defines the Get_Response subpage, which receives responses from the server side. The *getLibraryReply* transition retrieves a response from the USELIBRARY instruction - if the instruction was successful then the client will be added to the LIBRARY_SET state, otherwise the server is returned to IDLE. The *getCreateComponentReply* transition is responsible for determining whether or not a CREATECOMPONENT instruction was successful. The *getInitCompleteReply* places the client in the VISUALISATION_READY state if the server

replies with a success to the INIT_COMPLETE instruction, otherwise the server is returned to the LIBRARY_SET state. Responses to an instruction sent to the *EVP* are retrieved by the *getResponse* transition. The final transition, *getReply*, obtains a response from the EVP in response to an AWAITING instruction. This transition is distinct from the getResponse transition because the arguments of the S(uccess) response are inspected to see if the command is CONT or STOP.

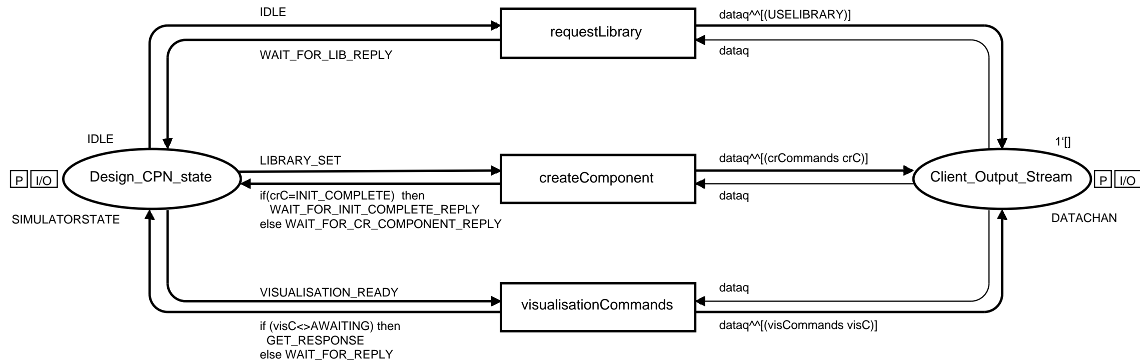

Fig. 10. Subpage for the Issue_Instruction transition.

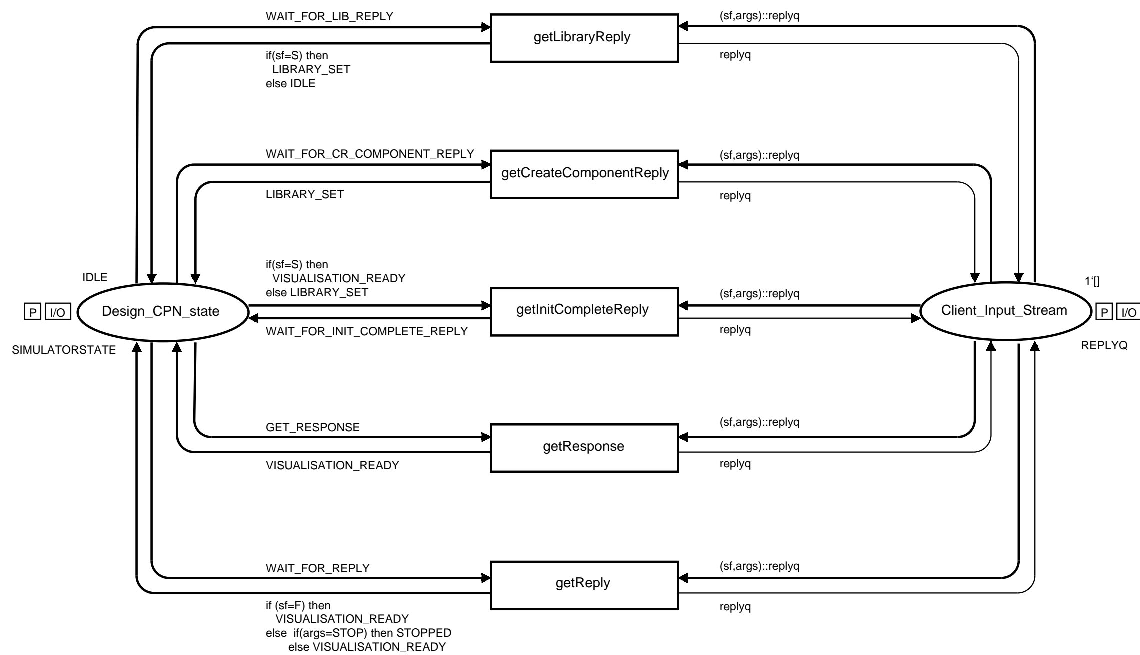

Fig. 11. Subpage for the Get_Response transition.

There are six transitions contained within the subpage of the Accept_Instruction substitution transition as shown in Figure 12. These transitions are activated when instructions arrive in the correct order. The *processUseLibraryInstruction* transition is enabled only when *EVP* is IDLE and a USELIBRARY instruction is received. The USELIBRARY instruction is removed from the queue and a *S* reply is issued to signify that the instruction was suc-

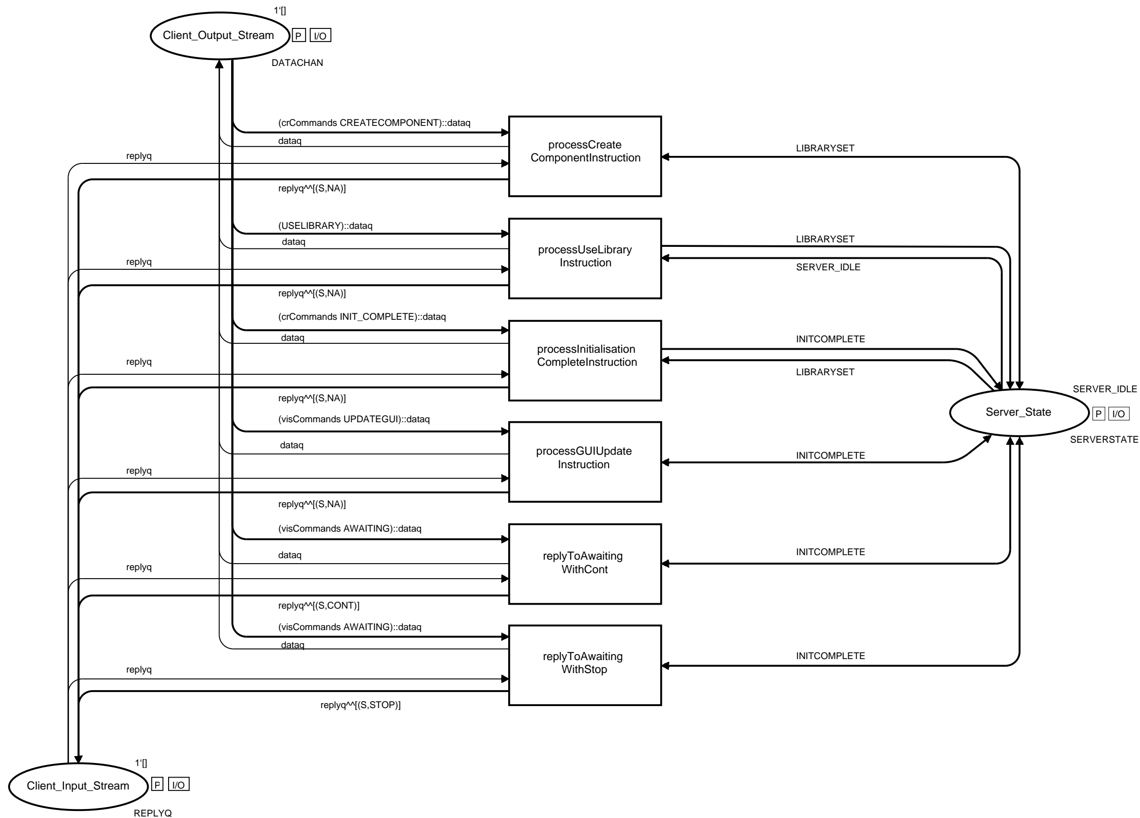

Fig. 12. Subpage for the Accept_Instruction transition.

cessful. The *processCreateComponentInstruction* transition is enabled only when the *EVP* is in the LIBRARYSET mode and a CREATECOMPONENT instruction is received. The CREATECOMPONENT instruction is removed from the queue and a *S* reply is issued. The *processInitialisationCompleteInstruction* transition is enabled only when the *EVP* is in the LIBRARYSET mode and an INIT_COMPLETE instruction is received. The instruction is removed from the queue and a *S* reply is issued, the *EVP* state is then changed to INITCOMPLETE. The *ProcessGUIUpdateInstruction* transition is enabled only when an UPDATEGUI instruction is received and the *EVP* is in the INITCOMPLETE mode. The UPDATEGUI instruction is removed from the queue and a *S* reply is issued. The *replyToAwaitingWithCont* transition is enabled when an AWAITING command is received and the server is in the INITCOMPLETE state. The AWAITING instruction is removed from the queue and a *(S,CONT)* reply is sent. The final transition, *replyToAwaitingWithStop*, is enabled when an AWAITING command is received and the server is in the INITCOMPLETE state. The AWAITING instruction is removed from the queue and a *(S,STOP)* reply is sent.

The four transitions contained within the Discard_Instruction substitution transition can be seen in Figure 13. These transitions are activated when an instruction arrives out of order and is hence discarded.

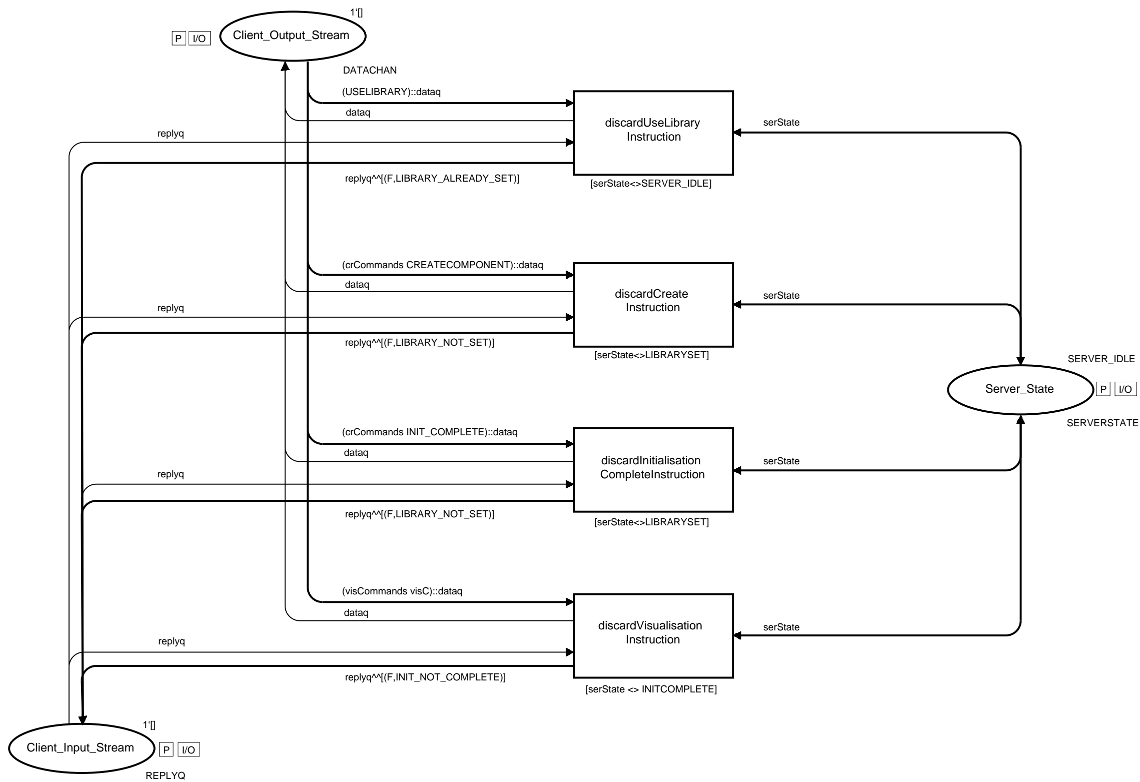

Fig. 13. Subpage for the Discard_Vis_Instruction transition.

8 Analysis of the Improved Visualisation Protocol

Figure 14 shows the reachability graph for the improved protocol with a queue length of one. As with the original protocol, there is only one terminal marking, which is at node 15. This node has similar characteristics to node 11 in Figure 6.

By examining the reachability graph we can observe the following:

- The occurence of the binding elements 1:1->2, 2:2->3, 3:3->4 shows a USELIBRARY instruction is sent and replied to sucessfully, this corresponds to the useLibaray part of the regular expression presented in Section 3.1.
- At node 4 two paths may be followed. By the occurrence of binding elements 5:4->6, 7:6->8 and 9:8->4, we can see that it is possible to send as many CREATE_COMPONENT instructions as we want with each being followed by a successful response. This path can be taken zero or more times. The only other path that can be taken from node 4 is by the occurrence of binding elements 4:4->5 6:5->7 and 8:7->9, which causes the sending of the INIT_COMPLETE command followed by a successful response from the server. These two possibilites correspond to the second and third terms of the regular expression - being createComponent* initComplete.
- The remaining biding elements show the sending and receiving responses to visualisation instructions, these correspond to the fourth term in the regular expression - [Visualisation Instructions]*.

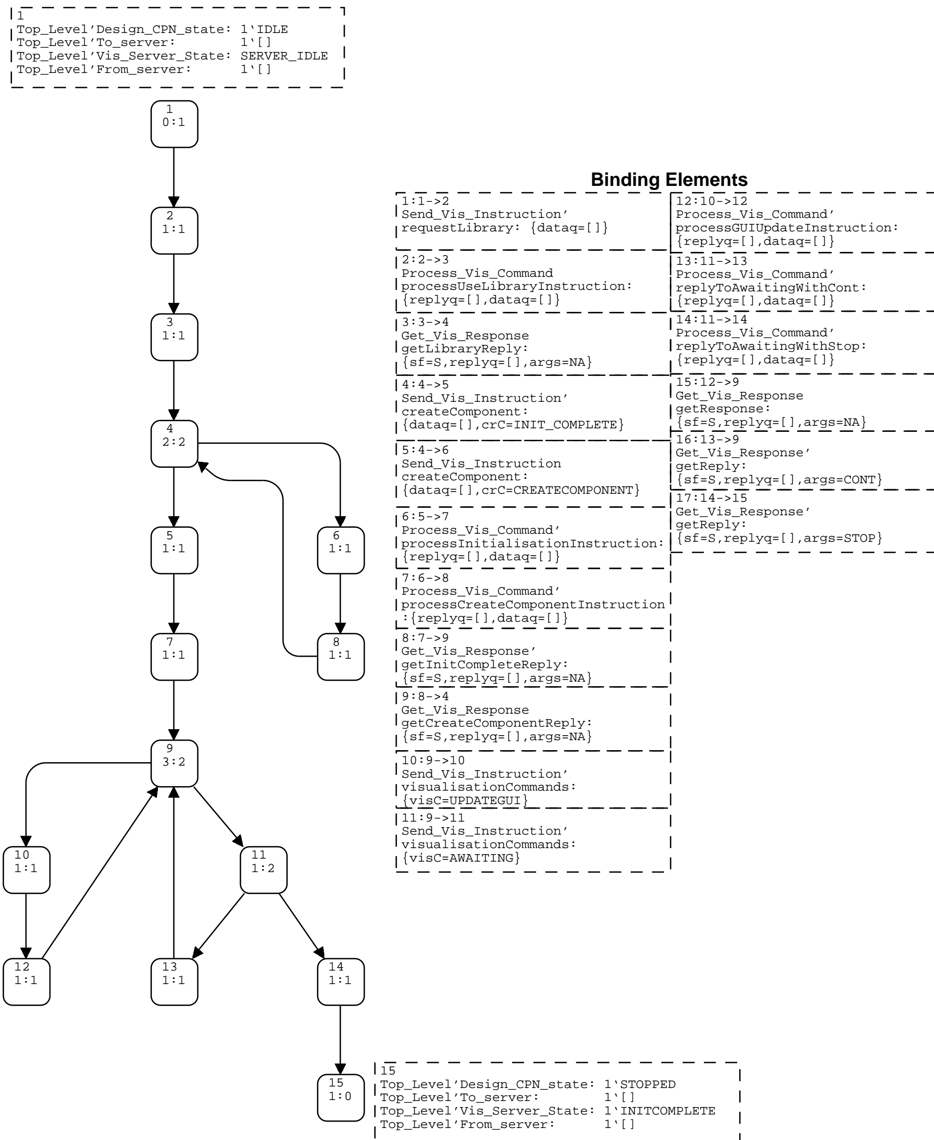

Fig. 14. Reachability Analysis for Model of Improved Protocol.

In summary, we can see that the client sends the instructions in the order specified by the regular expression (equation 1), and that they are also received by the server in the correct order, therefore we can say that the improved protocol behaves as expected.

Figure 15 shows the *Strongly Connected Component (SCC)* graph calculated from the CPN. As with the original protocol, the SCC graph contains less nodes than the reachability graph, indicating that there are cycles present. Also shown however is the fact that the only terminal node of the SCC contains the node that is the dead marking, node 15. This proves that there are no livelocks within the protocol.

Increasing the queue length beyond 1 has no effect as new instructions are not placed in the queue until the previous one is acknowledged. The only side effect of the new protocol is that it is reduced to a stop and wait protocol as after each instruction is sent to the *EVP* it waits for a response. This will lead to reduced performance on a network with high latency due to the fact that information has to travel in both directions whereas in the original protocol information was sent in only one direction a majority of the time.

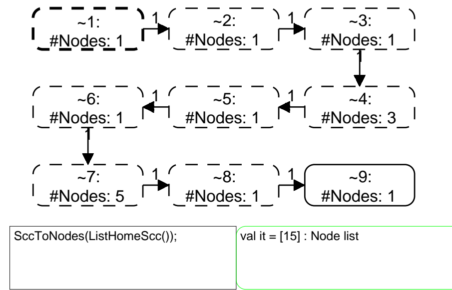

Fig. 15. SCC graph for the Improved Visualisation Protocol.

9 Conclusions

This paper shows that the visualisation protocol presented here is acceptable, if not perfect (due to the fact the sender of instructions is not informed of errors). The protocol allows a client application to request a remote a visualisation of system behaviour using the *External Visualisation Package*. It has been proved that the protocol is deadlock and livelock free and complies with the specification of the order in which instructions can be sent. It is therefore safe to use.

An important lesson learnt from this work was that the modelling and analysis exercise did reveal that the first cut implementation of the protocol (not modelled in this paper) did suffer from two problems. Firstly, that the order of instructions was not adhered to (any order was allowed) and that the implementation could deadlock. This has vindicated the use of formal methods in this project. The use of CPNs has proved to provide useful insight into the protocol's behaviour. This was achieved through several iterations of the model before the desired result was obtained.

Using CPNs, this paper has also specified an improved protocol, that returns information about the success of an instruction to the client. This will allow users of the protocol to provide for error handling. The improved protocol preserved the same desired properties as the original protocol. The drawback to feedback about the instruction sent is that the amount of data sent on the network is almost doubled as information is sent in both directions for each instruction, whereas in the original protocol, information flow was primarily in one direction. This is not seen as a problem for the envisaged application environment of a local area network.

It is also clear that the analysis of this practical application was well within the limits of Design/CPN's capabilities, as the state spaces remained very small. This has allowed much greater confidence to be gained in the design of this protocol and should encourage the use of these techniques in similar distributed systems projects.

Acknowledgments We would like to thank Guy Gallasch, for assistance in the original design of the visualisation protocol and for assistance with the layout of this paper. We would also like to thank members of the Air Operations Division of DSTO for their feedback on the Visualisation tool.

References

1. Animation by Mimic/CPN. `http://www.daimi.au.dk/designCPN/libs/mimic/`.
2. D. E. Comer. *Computer Networks and Internets*. Prentice-Hall International, Inc., 1997.

3. Mathew Elliot and Guy Gallasch. Final Year Project Report: Software Design Description. Technical report, University of South Australia, 2001.
4. Mathew Elliot and Guy Gallasch. Final Year Project Report: Software Project Management Plan. Technical report, University of South Australia, 2001.
5. G. Gallasch and L. M. Kristensen. The comms/CPN library. `http://www.daimi.au.dk/designCPN/libs/commscpn/`.
6. G. Gallash and L. M. Kristensen. Comms/CPN: A Communication Infrastructure for External Communication with Design/CPN. In *Proc. of 3rd Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN'01)*, pages 79–93. Department of Computer Science, University of Aarhus, 2001. DAIMI PB-554.
7. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volumes 1-3.* Monographs in Theoretical Computer Science. Springer-Verlag, 1997.
8. 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.
9. Kenneth C. Louden. *Compiler Construction: Principles and Practice.* PWS Publishing Company, 1997.
10. Design/CPN Message Sequence Charts library. `http://www.daimi.au.dk/designCPN/libs/mscharts/`.
11. Design/CPN Online. `http://www.daimi.au.dk/designCPN/`.