Modelling Layered Protocols in LOOPN

C.A. Lakos
Department of Computer Science
University of Tasmania,
Hobart, TAS, Australia.
C.A.Lakos@cs.utas.edu.au

C.D. Keen
Department of Computer Science
University of Tasmania,
Hobart, TAS, Australia.
C.D.Keen@cs.utas.edu.au

Abstract

LOOPN is a language and simulator for specifying systems in terms of coloured timed petri nets. It includes object-oriented features such as subtyping, inheritance and polymorphism which allow for the convenient modularisation of complex specifications. This paper briefly describes LOOPN and considers its application to the modelling of layered network protocols.

1 Introduction

Petri nets have been used for the modelling and simulation of concurrent systems since their introduction in the 1960s by C.A. Petri. Interest continues in them unabated (see [2], [6], [7] for example). The simplicity of the model makes automated analysis possible though such analysis suffers from the common problem of state space explosion. Recent work suggests that these limitations may be surmountable in some cases [20], [21], [22].

The modelling of large concurrent systems demands some form of modularisation to break down the complexity [18]. Recent interest in object-oriented programming has prompted the application of these concepts to the modularisation of petri nets [1]. LOOPN, a Language for Object-Oriented Petri Nets, is one approach to this synthesis of ideas.

The primary motivation for the development of LOOPN was the provision of an appropriate tool for teaching layered network protocols. Originally, a Protocol Workshop [5] was used, which displayed on a character screen the flow of messages associated with the simulation of Tannenbaum's Data Link protocols [19]. The major shortcoming of this tool was that the protocols were fixed leaving little or no room for experimentation. Initially a system was developed for the specification and simulation of petri nets. This was later extended to include object-oriented features so that more complex protocols could be demonstrated and investigated in an appropriate environment. We anticipate that these techniques will have desirable spinoffs for the analysis of petri nets.

This work has some points of contact with the earlier PROT nets [1] which had a more complete simulation package than we are currently presenting. However, it limited the object-orientation to the declaration of petri net modules and instances (without inheritance and without object-oriented declaration of token types); it had a proliferation of files (with some of a net description being given graphically and some textually); and the notation was rather cryptic (to simplify the mapping into the target language Pascal).

In this paper we present an overview of the current design of LOOPN and then consider its application to the modelling of layered network protocols.

2 Overview of LOOPN

The petri nets which are described by LOOPN are an extension of coloured nets with the provision of object-oriented features. As a simple running example, we consider the dining philosophers net of Fig 2.1 taken from [21].

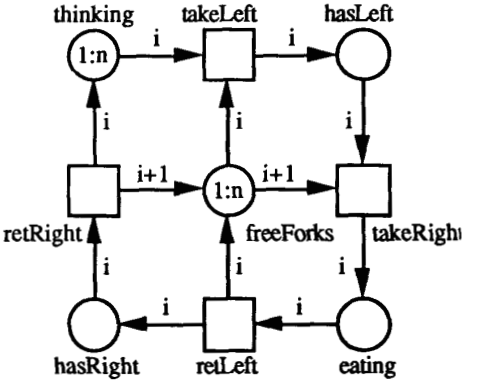

Fig 2.1 Dining philosophers petri net

The description of a petri net in LOOPN includes the declaration of constants, types, places, and transitions. Declarations generally follow the syntax of Pascal [9].

2.1 Type declarations

Type declarations include a subset of Pascal type declarations – in particular, enumerations, predefined types (integer, real, boolean, char), strings, single dimensioned arrays and records. We refer to these as **basic types**. Basic types can be defined in terms of other basic types in the usual way.

LOOPN also supports the declaration of **token types**. These are defined as extended record structures which encapsulate a set of data fields (each of some basic type) and a set of associated functions in a single object. The data fields determine the range of possible colour combinations that tokens of this type may assume.

A place holds a list of tokens of a specified token type. Each time a token is added to a place, it is timestamped according to the current simulation time. Suitable use of these time stamps allows LOOPN nets to be interpreted as timed or stochastic nets [7], [15].

In true object-oriented style, a token type is declared as a subtype of one or more other token types, and thus inherits the parents' data fields and functions. It may augment the features of the parent and may override the parent's defined functions. The most elementary token type, which all others ultimately inherit is called *null*. While it has no data fields, it includes definitions of the functions *first*, *last*, *delay*, which are therefore available to every other token type. These functions interrogate the token time stamps. The functions *first*, *last* are parameterless functions returning true if the specified token is the first, respectively last, token to arrive at the place. The function *delay(t)* returns true if the token has been resident at the place for time *t*. The notation for referencing such token functions is

tokenid . functionid (functionargs)

For example, the token type declaration for the dining philosophers net would be:

```
TYPE
   phil_num   =  1..n;
   phil_type  =
         TOKEN null WITH
            ph : phil_num ;
            next = phil_num : self.ph mod n + 1;
            has (p:phil_num) = EXISTS x: x.ph = p;
         END;
```

This declares a token type which is a subtype of *null*, with a field holding a philosopher number. The function *next* returns the number of the neighbouring philosopher. Note that the identifier *self* can be used to refer to the current token. The function *has* can be used to determine if a place contains a specified philosopher. The existential quantifier *EXISTS* ranges over all the tokens currently at a place.

2.2 Places

Places are declared to be of a specified token type, thus limiting the tokens which may be resident at that place. Place declarations may also specify an optional restriction which imposes a universal filter on the place. Only those tokens satisfying the restriction are visible to the net, whether for the purposes of firing transitions or for evaluating conditions on the tokens at the place. This universal filter is useful for localising or encapsulating the desired place behaviour, but it can be considered simply as syntactic sugar for the conjoining of this condition to every transition taking input tokens from the place.

For example, the place declaration below specifies that philosopher tokens must reside in the place eating for two simulated time units before being visible.

```
PLACE   eating : phil_type | self.delay (2);
```

If the selection criteria on a place does not depend on one of these time related functions then the selection ranges over all the tokens at the place.

2.3 Transitions

Transitions specify input places, output places, and auxiliary actions, each component being optional. (Unlike PROT nets [1], LOOPN has no restrictions on the number of input and output places of a given token type.) Each input token is explicitly named, together with the place from which it is derived. There is also an optional condition which the selected token must satisfy in order for the transition to fire. Such conditions are equivalent to the global restrictions applied to places, except that the restriction of token visibility is now local to the transition rather than global. Input conditions thus provide an additional level of filtering of tokens, as shown in Fig 2.2.

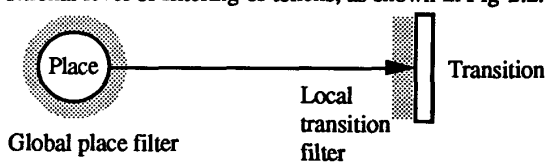

Fig 2.2 Filtering of tokens

The transition output places are also explicitly named, together with the tokens which are to be added to those places. All token identifiers have scope local to the transition. The token values may be copies of existing (input) tokens, copies of existing tokens with certain named fields changed, or newly generated tokens with values specified for named fields. The optional auxiliary action of a transition consist of one or more procedure calls. It does not affect the firing of the net, but allows interaction with the outside world. Typically it is used to report the progress of the simulation.

For example, the transition for a philosopher to pick up a right hand fork would be:

```
TRANSITION takeRight;
   INPUT   i <- hasLeft;
           j <- freeForks | j.ph = i.next;
   OUTPUT eating <- i;
```

2.4 Modules and module instances

A petri net in LOOPN is coded as one or more modules, each of which consists (in the simplest case) of constant, type, place, and transition declarations. Immediately preceding the transitions is the initialisation for the module. Syntactically, this is similar to a transition with no inputs specified. Semantically, it is executed once at startup to establish the initial petri net marking of the module.

For example, the module for the dining philosophers would be:

```
MODULE philosophers (CONST which: integer);
   CONST   n = 5;
   TYPE    phil_num = 1..n;
           phil_type = ... ;
   PLACE   thinking, eating, freeForks,
           hasLeft, hasRight : phil_type;
   INITIALISATION;
      OUTPUT thinking <- i1 = [ph:1];
             thinking <- i2 = [ph:2];
                 ...
   TRANSITION takeLeft;
      INPUT   i <- thinking;
              j <- freeForks | j.ph = i.ph;
      OUTPUT hasLeft <- i;
      ACTION printf ("Philosopher %d has raised
                      her left fork", i.ph);
   ...
   ACCESS is_thinking (p:phil_num) =
                      boolean: thinking.has (p);
          is_eating (p:phil_num) =
                      boolean: eating.has (p);
END MODULE
```

Complex petri nets may be built up by including in a module instances of other modules. In the Eiffel terminology, a module may be a client of other modules [16]. For example, from dining philosophers module above, we could build a module for a lodge containing three rooms of dining philosophers by including three instances:

```
MODULE lodge;
   CONST   first = 1;  second = 2;   third = 3;
   INITIALISATION;  { No additional initialisation }
   INSTANCE firstRoom : philosophers (first);
   INSTANCE secondRoom: philosophers (second);
   INSTANCE thirdRoom : philosophers (third);
END MODULE
```

The declaration of an instance effectively duplicates the associated subnet, including its places, transitions and nested instances. It is therefore not permissible to instantiate modules recursively, i.e. within module *M* it is not permissible to instantiate *M*.

Modules can interact with their environment in a number of ways. At the simplest level, they may include constant parameters, so that an instance can determine its identity (out of a number of instances). Thus the above dining philosophers module has a constant integer parameter so that the module code can determine the particular room of dining philosophers to which it pertains.

Secondly, a module may interact with its environment by declaring parameter places. Here the interaction with the environment is by transferring tokens into and out of the module. Because of this interface to places, modules may be thought of as *super transitions* (as opposed to *super places* [18], the latter not being provided in LOOPN). Parameter places may be declared with usage *INPUT*, *OUTPUT* or *INOUT* (i.e. input and output). Only those parameters with *INPUT* or *INOUT* usage may have a restriction attached, in which case the tokens in that place are only visible within the module if that condition is satisfied in addition to other global visibility restrictions.

A third form of interaction is to provide access functions, which support the querying of the module state without the need to transfer tokens. In the dining philosophers module, we provided access functions to determine whether a given philosopher is eating or thinking. The access functions of a module instance may be referenced in the usual fashion:

```
instanceid . functionid ( functionargs )
```

In order to support flexible access to a module's access functions, it is necessary to allow module instances to be parameters to modules.

LOOPN supports the definition of subclasses of modules, with inheritance. A module can be defined to be a subtype of another module and inherit all of the features of its parent module type. (Currently only single inheritance is implemented but multiple inheritance is being considered.) A module subtype can augment the features of its parent and can override declared identifiers of the parent. Polymorphism of modules is supported by allowing an instance of a module subtype to be used where an instance of a parent module type is specified. In this way, complex petri nets may be built by including instances of other modules, and by augmenting or modifying existing modules.

For example,we could extend the dining philosophers module to allow each philosopher to carry a book, which may be exchanged while the philosopher is thinking. This module would be defined:

```
MODULE learned_philosophers = philosophers;
  CONST  m = 100;          { 100 books available }
  TYPE book_num   =  0..m; { Book 0 for no book }
       book_type  =  TOKEN null WITH
                       bk : book_num;
                     END;
       new_phil   =  TOKEN phil_type, book_type
                     END;
  PLACE  thinking, eating, freeForks,
         hasLeft, hasRight : new_phil;
         freeBooks : book_type;
  INITIALISATION;
    OUTPUT thinking <- i1 = [ph:1, bk:0];
           thinking <- i2 = [ph:2, bk:0];
           ...
           { Set up library - use subnet? }
           freeBooks <- b1 = [bk:1];
           freeBooks <- b2 = [bk:2];
           ...
  TRANSITION changeBook;
    INPUT  i <- thinking;
           j <- books;
    OUTPUT thinking <- i2 = i [bk:j.bk];
           books <- j2 = [bk:i.bk];
END MODULE
```

Note that the first line indicates that the module *learned_philosophers* inherits the features of the parent module *philosophers*, including the subsequent changes. The most significant change is to alter the type of the places so as to hold tokens of the *new_phil* subtype. The inherited transitions are still applicable, and the transfer of tokens will guarantee that the philosophers will not lose their books in the process of eating.

2.5 Object-oriented features

The brief description of LOOPN above shows that it provides a powerful and flexible notation for specifying large, complex concurrent systems. One aspect of this flexibility is the variety of ways a module may be encapsulated and interact with the rest of the net. It may be self-contained, with only constant parameters to identify the particular instantiation. It may provide status information to other modules through access functions. Or it may transfer tokens via interface places.

Another aspect of this flexibility is the variety of ways that token visibility can be controlled. At the global level, a place can have a restriction which affects the visibility of all tokens at the place. In true object-oriented style, this allows the place to control or specify its own token strategy. Thus we can define places which act as stacks, queues, or even priority queues. At the local level, each transition can decide which tokens are appropriate to its own purposes, in addition to the global restrictions imposed by the place. At an intermediate level, each module can specify which tokens are relevant, by specifying a restriction at the module boundary. We depict this three-stage filtering in Fig 2.3.

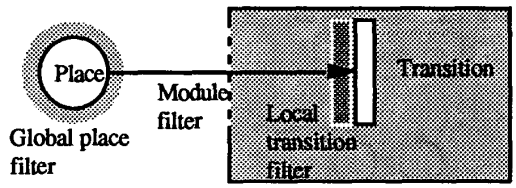

Fig 2.3 Filtering of tokens with modules

LOOPN supports inheritance of token types, so that one type inherits the features of one or more parents, and then may augment those features, and override selected functions. With this inheritance comes polymorphism, since a token of some subtype may be used where tokens of a parent type are expected. Thus, a place of a given token type may contain tokens of some subtype. Similarly, a module declared with a formal place parameter of some token type may be bound to an actual place containing tokens of some subtype.

LOOPN also supports module inheritance. A module inherits the features of a parent, and may augment those features and override selected places, transitions, instances and access functions. Polymorphism of modules is supported by allowing instances of some module subtype to be used where instances of some parent module type is specified. In this way, complex petri nets may be built by including instances of other modules, or by augmenting or modifying existing modules.

3 Modelling layered protocols

We now consider the application of LOOPN to the modelling of layered network protocols. The particular scenario we consider is a three-layer protocol. Layer 1 is the physical channel which may transfer, lose or corrupt messages. Layer 2 is the data link layer, which recovers from the transmission errors of layer 1, and also includes pipelining for better channel utilisation. Layer 3 is some simple protocol component which transfers text messages assuming the error-free transmission of layer 2. We do not consider layer 3 further since it would add little to the discussion.

3.1 Overall structure and interaction

We commence by considering the overall structure of the layers and their interaction. An appropriate structure is considered to be that shown in Fig 3.1. Each protocol component is a separate module with a number of interface places, which serve as intermediate transfer points for the messages. In the more precise terminology of the ISO

protocols, they are service access points [19]. We have labelled these interface places with names indicating the layer below, the use of the place input or output to that, and a letter indicating the side of the protocol stack.

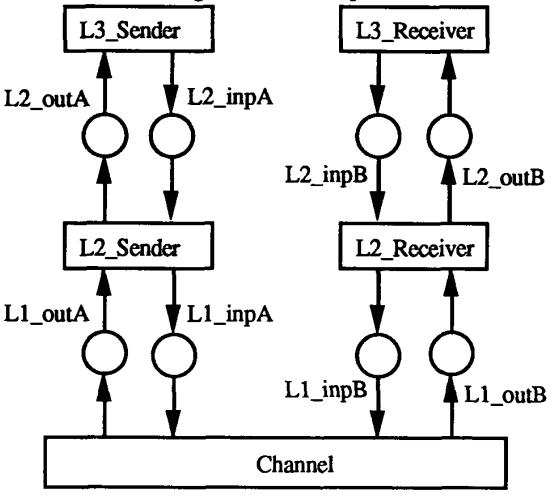

Fig 3.1 Overall interaction structure for the layered protocols

One of the key issues in the overall design is the appropriate token types to be used at each level. We can initially assume one token type for use within each layer (called *L1_type*, *L2_type*, and *L3_type*), and one token type for each level of interface (called *L1_2_type* and *L2_3_type*). Initially, we assume that since layer 3 is concerned with the transmission of text strings, *L3_type* will include a text field and functions for accessing that text field. Since layer 2 is concerned with recovery from transmission errors, *L2_type* will need to store a message kind field, a sequence number field, and a checksum, together with appropriate functions. Finally, since layer 1 must deal with the transmission, loss and corruption of messages, *L1_type* will at least need to include a checksum field and associated functions. As a result of this, we arrive at the overall module structure of Fig 3.2.

In the following subsections we will consider the issues pertinent to the refinement of the overall design above. This will cover the protocols in layers 1 and 2. As noted above, a detailed consideration of the layer 3 protocol would add little to the discussion and so we omit it. It is sufficient for our purposes to assume that it transfers textual messages.

3.2 Channel protocol – information hiding

In order to refine the token types and the interactions given above, we first consider the petri net modules which would be appropriate for the level 1 or channel protocol.

```
MODULE mesg_types;
 TYPE pkt_kind =  (DATA, ACKN);
      seq_type =  (0..7);
      L1_type  =  TOKEN ... WITH
                    chkOK: boolean;
                    ...
                  END;
      L2_type  =  TOKEN ... WITH
                    kind:  pkt_kind;
                    seq:   seq_type;
                    chkOK: boolean;
                  END;
      L3_type  =  TOKEN ... WITH
                    mesg: STRING;
                  END;
      L1_2_type = TOKEN ... WITH ... END;
      L2_3_type = TOKEN ... WITH ... END;
  INITIALISATION;
END MODULE

MODULE L1_protocol = mesg_types (
     INPUT    L1_inpA, L1_inpB: L1_type;
     OUTPUT   L1_outA, L1_outB: L1_type);
  ...
END MODULE

MODULE L2_protocol = mesg_types (
     INPUT    L2_inp, L1_inp: L2_type;
     OUTPUT   L2_out, L1_out: L2_type);
  ...
END MODULE

MODULE L3_protocol = mesg_types (
     INPUT    L2_inp: L3_type;
     OUTPUT   L2_out: L3_type);
  ...
END MODULE

MODULE driver = mesg_types;
  PLACE  L1_inpA, L1_inpB,
         L1_outA, L1_outB : L1_2_type;
         L2_inpA, L2_inpB,
         L2_outA, L2_outB : L2_3_type;
  INSTANCE
    L1_channel : L1_protocol
         (L1_inpA,L1_inpB,L1_outA,L1_outB);
    L2_sender : L2_protocol
         (L2_inpA,L1_inpA,L2_outA,L1_outA);
    L2_receiver : L2_protocol
         (L2_inpB,L1_inpB,L2_outB,L1_outB);
    L3_sender : L3_protocol (L2_inpA,L2_outA);
    L3_receiver : L3_protocol (L2_inpB,L2_outB);
END MODULE
```

Fig 3.2 Overall module structure of the layered protocol

The simplest construct would be an error-free or good channel, the unidirectional version being presented in Fig 3.3 and Fig 3.4.

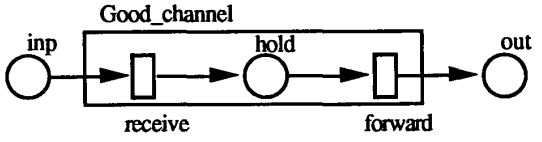

Fig 3.3 Interaction for a good, error-free channel

```
MODULE good_channel = mesg_types (
      INPUT inp:null; OUTPUT out:null);
   PLACE   hold: null | self.delay (2);
   INITIALISATION
   TRANS receive;
      INPUT    mg <- inp;
      OUTPUT hold <- mg;
   TRANS forward;
      INPUT    mg <- hold;
      OUTPUT out <- mg;
END MODULE
```

Fig 3.4 Module definition for a good, error-free channel

The module introduces a delay of two time units for the transmission of tokens from the input to the output place. We have chosen the token type *null* for the formal place parameters, since the above channel module has no need for any additional information. But since *null* is a parent type for all token types, polymorphism allows the above module to be used for transferring tokens of any type. The explicit naming of tokens in the transitions allows us to analyse the module to determine the proper transfer of tokens and the consistent subtyping at the module interface places. This approach to token type specification is highly desirable since syntactic checks will guarantee that no other information in the token is visible within the module, and hence cannot affect the firing of transitions, thus simplifying analysis of the net.

The above module can now be instantiated twice to form a bidirectional channel for layer 1 of our protocol stack. It could also be used without modification to transfer messages directly between the layer 3 protocol components – either to illustrate peer-to-peer protocols, or to debug one protocol layer at a time.

The good channel may be extended to one which loses messages (a lossy channel) as in Fig 3.5 and Fig 3.6.

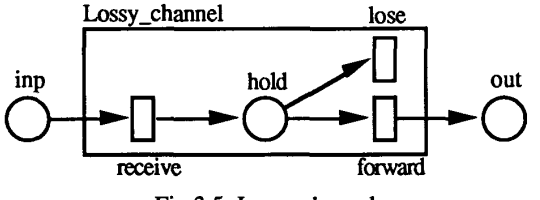

Fig 3.5 Lossy channel

```
MODULE lossy_channel = good_channel;
   INITIALISATION;
   TRANS lose;
      INPUT mg <- hold |
            randomreal(globalseed) <= 0.001;
END MODULE
```

Fig 3.6 Module definition for lossy channel

This module is simple enough to be written from scratch, but it is instructive to see (even in this example) how a more complex module may be derived from a simpler one. Thus a lossy channel is similar to a good channel, except with an additional transition, which has an attached condition ensuring that the probability of message loss is at most 0.001.

The reader will note that this channel still operates on *null* tokens, and therefore can be used to transfer any tokens. However, the possibility of message loss means that the channel is no longer appropriate for illustrating or debugging peer-to-peer protocols at higher layers.

The final stage in the development of the channel protocol is to introduce the possible corruption of messages. Now the tokens transferred must include some checksum indication which we have already included in the definition of the token type *L1_type*. The resultant petri net is illustrated in Fig 3.7 and Fig 3.8.

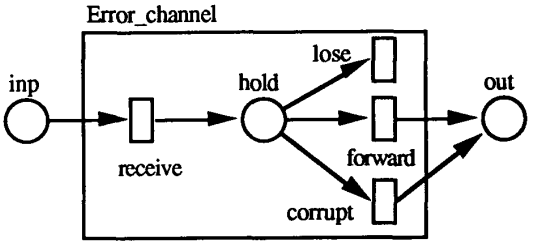

Fig 3.7 Diagram of an error channel

```
MODULE error_channel = lossy_channel (
      INPUT  inp: L1_type; OUTPUT out: L1_type);
   PLACE  hold : L1_type | self.delay(2);
   INITIALISATION;
   TRANS corrupt;
      INPUT     mg1 <- hold |
                  randomreal(globalseed) < 0.05
      OUTPUT  out <- mg2 = mg1 [chkOK:FALSE];
END MODULE
```

Fig 3.8 Module definition of an error channel

Once again, we could have developed this module from scratch, but we chose to illustrate the development from an earlier module. Note that we have made use of the facility of overriding a place (whether local or parameter) by a place with some token subtype. The other transitions for message transfer (without corruption) and for message loss are inherited from the parent module.

3.3 Layer 2 protocol and token subtyping

Tannenbaum [19], in his presentation of a typical data link layer protocol, gives a number of protocols, each of increasing complexity. However, each version is presented in its entirety, and the reader is left to deduce the changes made from one version to the next. With object-oriented nets, the changes can be explicitly indicated, thus improving comprehension.

The simplest form of layer 2 protocol is a stop-and-wait protocol. This caters for different speeds of sender and receiver but not for message loss. The one-directional protocol components are given below in Fig 3.9 and Fig 3.10.

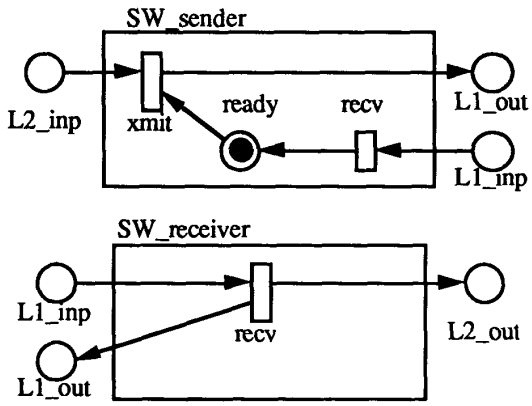

Fig 3.9 Stop and wait protocol sender and receiver

```
MODULE SW_sender = mesg_types (
      INPUT   L2_inp, L1_inp: null;
      OUTPUT  L1_out: null);
   PLACE ready: null;
   INITIALISATION;
      OUTPUT ready <- tok = [];
   TRANS xmit;
      INPUT   mg <- L2_inp;
              ok <- ready;
      OUTPUT  L1_out <- mg;
   TRANS recv;
      INPUT   ok <- L1_inp;
      OUTPUT  ready <- ok;
END MODULE

MODULE SW_receiver = mesg_types (
      INPUT   L1_inp: null;
      OUTPUT  L2_out, L1_out: null);
   INITIALISATION;
   TRANS recv;
      INPUT   mg <- L1_inp;
      OUTPUT  L2_out <- mg;
              L1_out <- ok = [];
END MODULE
```

Fig 3.10 Modules for stop-and-wait protocol

As with the initial channel protocols, we can specify the stop-and-wait protocol in terms of tokens of type *null*. This means that the protocol components can be used to transfer messages of any token type.

When we consider the possible loss and corruption of messages by the channel, the token type for the data link layer, namely *L2_type*, needs to include a sequence number field, a packet kind field (to distinguish data messages and acknowledgements) and a checksum field (to indicate whether the message is corrupted or not). This information originates within the layer and is appended to messages which pass through the layer. Therefore, the types of tokens on input and output to the module will differ, and we can no longer simply pass tokens of some subtype through the protocol layer.

This issue can be considered more closely by focussing on the basic process for forwarding a token, namely a transition of the form:

```
TRANS xfer;
   INPUT   tok <- inp_plc;
   OUTPUT  out_plc <- tok;
```

Here, the token *tok* must have a type which is the same as, or a subtype of, the token types of *inp_plc* and *out_plc*. Since it is a simple matter to track the transfer of tokens through a net it is possible to check consistency at the interface places of the module. However, as soon as we wish to modify the token by setting a sequence number field, the transition will need to be of the form:

```
TRANS xfer;
   INPUT   tok <- inp_plc;
   OUTPUT  out_plc <- tok2 = tok [seq:val];
```

This will add to the place *out_plc* a new token called *tok2*, which is a modified copy of the token *tok*. If the type of *inp_plc* already contains a *seq* field, the situation is as before. If, however, the type of *inp_plc* does **not** include the field *seq* then we require a *seq* field to be added. The matter is further complicated by the fact that polymorphism means that the type of token *tok* may be a subtype of the declared type for place *inp_plc*. This subtype will have hidden features, and we require these to be faithfully transmitted as well.

More precisely, if we write *x.type* for the token type associated with place or token *x*, and *t1+t2* for the token type *t1* augmented by the features of the token type *t2*, then we can indicate that type of *tok* is a subtype of the token type of *inp_plc* by writing:

tok.type = inp_plc.type + hidden.type

where *hidden.type* embodies the hidden features of *tok*. From this, we insist that the type of *tok2* be determined from the type hierarchy and be given by:

tok2.type = out_plc.type + hidden.type

This treatment was inspired by the presentation in [3], which dealt with the transmission of hidden features in a functional language.

Following these considerations, we can now finalise the types of module *mesg_types* required for the protocol layers as in Fig 3.11, and we can present the resultant one-bit sliding window protocol of Tannenbaum [19] as in Fig 3.12 and Fig 3.13.

```
TYPE
   L1_type =   TOKEN null WITH
                  chkOK: boolean;
               END;
   L2_type =   TOKEN L1_type WITH
                  kind:  pkt_kind;
                  seq:   seq_type;
               END;
   L3_type =   TOKEN null WITH
                  mesg: text;
               END;
   L2_3_type = TOKEN L3_type END;
   L1_2_type = TOKEN L2_type, L3_type END;
```

Fig 3.11 Modified type structure for one-bit sliding window protocol

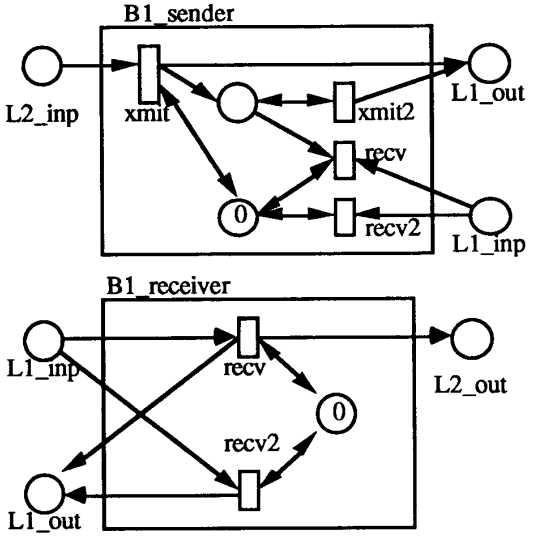

Fig 3.12 Interaction for the one-bit sliding window protocol

```
MODULE B1_sender = SW_sender (
      INPUT  L2_inp: null;
      INPUT  L1_inp: L2_type;
      OUTPUT L1_out: L2_type);
   TYPE
      window =
         TOKEN null WITH
            seq: seq_type;
            lim: seq_type;
            OK = boolean: self.seq = self.lim;
            next_seq = seq_type: (self.seq + 1) MOD 2;
            prev_seq = seq_type: (self.seq + 1) MOD 2;
            next_lim =  seq_type: (self.lim + 1) MOD 2;
            match (sq:seq_type) = boolean: self.lim =sq;
         END;
   PLACE    ready : window;
            buffer : L2_type;
   INITIALISATION;
      OUTPUT ready <- sq = [seq:0, lim:0];
   TRANS xmit;
      INPUT    mg <- L2_inp;
               rd <- ready | rd.OK;
      OUTPUT L1_out <- mg2 =
            mg [kind:DATA, seq:rd.seq, chkOK:TRUE];
               buffer <- mg2;
               ready <- rd2 = rd [seq:rd.next_seq];
   TRANS xmit2;
      INPUT    mg <- buffer | mg.delay(timeout);
      OUTPUT buffer <- mg;
               L1_out <- mg;
   TRANS recv;
      INPUT    ack <- L1_inp | ack.chkOK;
               rdy <- ready  | rdy.match(ack.seq);
               mes <- buffer | rdy.match(mes.seq);
      OUTPUT ready <- rdy2 = rdy [lim:rdy.next_lim];
   TRANS recv2;
      INPUT    ack <- L1_inp;
               rdy <- ready | NOT ack.chkOK OR
                              NOT rdy.match(ack.seq);
      OUTPUT ready <- rdy;
END MODULE

MODULE B1_receiver = SW_receiver (
      INPUT  L1_inp: L2_type;
      OUTPUT L2_out: null;
      OUTPUT L1_out: L2_type);
   TYPE window = TOKEN ... END;
                  { Same definition as above }
   PLACE expect : window;
   INITIALISATION;
      OUTPUT expect <- sq = [seq:0, lim:0];
   TRANS recv;
      INPUT    mg <- L1_inp | mg.chkOK;
               sq <- expect | sq.match(mg.seq);
      OUTPUT L2_out <- mg;
               L1_out <- mg2 = [kind:ACKN, seq:sq.seq,
                                chkOK:TRUE];
               expect <- sq2 = sq [seq:sq.next_seq,
                                   lim:sq.next_lim];
   TRANS recv2;
      INPUT    mg <- L1_inp;
               sq <- expect | NOT mg.chkOK OR
                              NOT sq.match(mg.seq);
      OUTPUT expect <- sq;
               L1_out <- mg2 = [kind:ACKN,
                        seq:sq.prev_seq, chkOK:TRUE];
END MODULE
```

Fig 3.13 Modules for one bit sliding window protocol

Note that there is a pleasing symmetry between the multiple possibilities of correct and incorrect message transmission and the multiple transitions in sender and receiver for correct and incorrect reception. Note also that we have chosen to retain the sequence number of the sender within the sender. This could have been regenerated from the received acknowledgement, but the flow of control would not have been as clear.

The next step in our development of data link protocols is to modify the above protocol to include pipelining. Pipelining allows multiple messages to be in transit at any time so that the channel utilisation can be increased. The normal approach (as in Tannenbaum) is to rewrite the protocol introducing an explicit sliding window and modifying the removal of messages from the buffer. However, by following the usual object-oriented guidelines of defining functions to perform relevant operations in our earlier protocols, we can now achieve the pipelining effect by simply redefining the behaviour of the window type, as shown in Fig 3.14.

```
MODULE B3_sender = B1_sender (
      INPUT   L2_inp: null;
      INPUT   L1_inp: L2_type;
      OUTPUT  L1_out: L2_type);
   TYPE
      window2 =
         TOKEN window WITH
            next_seq = seq_type: (self.seq + 1) MOD 8;
            prev_seq = seq_type: (self.seq + 7) MOD 8;
            next_lim = seq_type: (self.lim + 1) MOD 8
         END;
   PLACE   ready : window2;
   INITIALISATION;
      OUTPUT ready <- rd = [seq:0, lim:0];
END MODULE

MODULE B3_receiver = B1_receiver (
      INPUT   L1_inp: L2_type;
      OUTPUT  L2_out: null;
      OUTPUT  L1_out: L2_type);
   TYPE window2 = TOKEN ... END;
   PLACE expect : window2;
   INITIALISATION;
      OUTPUT expect <- sq = [seq:0, lim:0];
END MODULE
```

Fig 3.14 Sliding window protocol

It is left as an exercise to the reader to combine a layer 2 sender and receiver to form a two-way protocol component. Similarly, the reader may consider what is involved in changing the protocol so that messages are not individually acknowledged, i.e. the one acknowledgement can acknowledge all messages up to and including the specified sequence number.

4 Conclusions and further work

This paper has presented the design of the language LOOPN. We have shown that the language extends the notions of coloured petri nets with the addition of object-oriented features. These features include the definition of token types as classes, with the attendant inheritance and polymorphism facilities. Similarly, LOOPN modules can be defined as classes with inheritance and polymorphism.

We believe that LOOPN is well-suited to the concise modelling of complex concurrent processes. The language is compact without the need to provide complex control structures. It also encourages a declarative style of programming, since each transition operates independently and simply specifies the local conditions under which it fires. The implementation is straightforward and efficient, based on event-oriented simulation techniques.

The object-oriented features of LOOPN encourage the reuse of existing petri net modules. The flexible module structure allows the interface to be as broad or as narrow as desired, whether in terms of access functions or places.

Using layered network protocols as a case study, we have demonstrated how the above object-oriented features can facilitate the protocol design. Token subtyping and polymorphism encourage the production of protocols where, within one layer, the data from higher layers is transparent. This is the ideal form of information hiding in the context of layered protocols.

Similarly, we have demonstrated that LOOPN encourages the development of more complex protocols from simpler ones. As a result, minor changes in protocol specification result in minor changes in the associated modules. This is a property which Meyer refers to as continuity [16]. This is ideal for teaching purposes, and we also expect benefits in the analysis of the petri nets.

LOOPN has been implemented on Unix workstations at the University of Tasmania. It has been effective in allowing students to simulate significant systems without the problems of deadlock and concurrent access. They have also been able to experiment with layered protocols, building a layer 3 protocol in the context of the above stack.

LOOPN has also been used not just for simulating network protocols but for running them in realistic situations. Thus, we can experiment with protocols in the protected simulation environment and, with a minimum of change, transport them into a functioning network. LOOPN should therefore serve for prototyping as well as simulation modelling.

Currently, we are developing graphical user interfaces to LOOPN in contrast to the textual form discussed in the paper, but like the textual form, it will be complete in itself. One version is Macintosh-based while the other

is based on XWindows. The interesting aspect of this work is the representation of inheritance in a graphical way that provides meaningful information to the user but without cluttering the screen.

The highest priority for future work is to extend the semantics of coloured nets (as in [11]) to encompass inheritance. A key aspect to this will be a careful consideration of the restrictions which should limit overriding. The Eiffel approach ([16]) is to require that a function which overrides that of a parent must satisfy the parent's pre- and post-conditions. This guarantees that the derived class has a similar behaviour to the parent. We believe that similar restrictions ought to apply to LOOPN. The chief benefit would be proofs of correctness and the reuse of reachability analysis results. For example, we believe that the stubborn set reachability analysis of the dining philosophers net [21] should carry over with minimal change to our learned philosophers module, since the possession of a book by a philosopher does not affect the firing of any of the original transitions. The syntactically hidden information should give us directly a set of equivalent markings.

6 References

[1] M. Baldassari and G. Bruno, "An Environment for Object-Oriented Conceptual Programming Based on PROT Nets", *Advances in Petri Nets 1988, Lecture Notes in Computer Science 340*, Springer Verlag.

[2] E. Best, "DEMON: Design Methods Based on Nets", *Advances in Petri Nets 1989, Lecture Notes in Computer Science 424*, Springer Verlag.

[3] L.Cardelli and J.C.Mitchell, "Operations on Records", *Mathematical Foundations of Programming Semantics, Lecture Notes in Computer Science 442*, Springer Verlag, 1989.

[4] D. Comer, "Internetworking with TCP/IP: Principles, Protocols, and Architectures", Prentice-Hall, 1988.

[5] J. Colville, "Demonstration of Data Communication Protocols" *Technical Report 85.8*, School of Computing Sciences, New South Wales Institute of Technology, 1985.

[6] Th. Hildebrand and N.Trèves, "S-CORT®: A Method for the Development of Electronic Payment Systems", *Advances in Petri Nets 1989, Lecture Notes in Computer Science 424*, Springer Verlag.

[7] H.P. Hillion, "Timed Petri Nets and Application to Multi-Stage Production Systems", *Advances in Petri Nets 1989, Lecture Notes in Computer Science 424*, Springer-Verlag.

[8] G.K. Hutchinson, "Introduction to the Use of Activity Cycles as a Basis for Systems Decomposition and Simulation" *SImuletter, ACM SIGSIM*, 7,1 pp35-51, 1975.

[9] K. Jensen, and N. Wirth, "Pascal User Manual and Report", Springer-Verlag, 1975.

[10] K. Jensen, "Coloured Petri Nets", *Advances in Petri Nets 1986, Lecture Notes in Computer Science 254*, Springer-Verlag.

[11] K. Jensen, "Coloured Petri nets: A high level language for system design and analysis", *Advances in Petri Nets 1990, Lecture Notes in Computer Science 483*, Springer Verlag.

[12] B.W. Kernighan and D.M. Ritchie, "The C Programming Language", *Prentice-Hall Software Series*, Prentice Hall, 1978.

[13] W. Kreutzer, "System Simulation: Programming Styles and Languages", Addison-Wesley, 1986.

[14] C.A. Lakos, "Petsi: A Petri Net Simulator", *Technical Report 88-14*, Department of Computer Science, University of Tasmania, 1988

[15] M.A. Marsan, "Stochastic Petri Nets: An Elementary Introduction", *Advances in Petri Nets 1989, Lecture Notes in Computer Science 424*, Springer-Verlag.

[16] B. Meyer, "Object-oriented Software Construction", Prentice Hall, 1988.

[17] W. Reisig, "Petri Nets, An Introduction", Springer-Verlag, 1985.

[18] W. Reisig, "Petri Nets in Software Engineering" *Advances in Petri Nets 1986, Lecture Notes in Computer Science 254*, Springer-Verlag.

[19] A.S. Tannenbaum, "Computer Networks", Prentice-Hall, 1981.

[20] A. Valmari, "Stubborn Sets for Reduced State Space Generation" *Proceedings of 10th International Conference on Application and Theory of Petri Nets*, Bonn II, pp 1-22, 1989.

[21] A. Valmari, "Stubborn Sets fof Coloured Petri Nets", *Proceedings of 12th International Conference on the Application and Theory of Petri Nets*, Aarhus, 1991.

[22] G. Wheeler, A. Valmari and J. Billington, "Baby TORAS Eats Philosophers but thinks about Solitaire", *Proceedings 5th Australian Software Engineering Conference*, Sydney, pp 283-288, 1990.