Modelling Network Protocols with Object Petri Nets

Charles Lakos, John Lamp, Chris Keen, Brian Marriott
Computer Science Department
University of Tasmania
GPO Box 252C, Hobart, TAS, 7001
Australia.
{C.A.Lakos, J.W.Lamp, C.D.Keen, B.W.Marriott}@cs.utas.edu.au

Abstract: This paper examines how object-oriented extensions to the Petri Net formalism can address a number of issues in the modelling of network protocols. The object-oriented extensions lead to the formalism of Object Petri Nets, with a textual language form referred to as LOOPN++. The paper considers practical examples for which clean, well-structured models can be created because of the support for modularity, inheritance, polymorphism, genericity, and mobile objects.

1 Introduction

For a long time, computer network protocols have motivated research in concurrent systems. Their increasing complexity has fuelled ever-increasing budgets and human resource demands for developing and maintaining the associated software. This has provided an impetus for flexible software development environments capable of handling concurrent systems and maximising software reuse; and for formal techniques which can ensure software reliability.

Petri Nets have been one formalism which has been applied to network protocols with beneficial results [5, 6, 8, 16]. This has been facilitated by a number of attributes traditionally associated with Petri Nets – their formal definition, the associated executable models, and the amenability to automated analysis.

This paper addresses one area which has previously been identified as a weakness in Petri Net formalisms: *the absence of compositionality has been one of the main critiques raised against Petri net models* [12]. We propose a modified Coloured Petri Net (CPN) formalism [13] called Object Petri Nets (OPNs) [19, 20] which, as the name suggests, incorporates object-oriented structuring into Petri Nets. The goal of this formalism is to reap many of the benefits associated with object-oriented technology, such as more flexible and powerful structuring primitives and the practical support for software reuse.

The paper presents LOOPN++, a textual form of the OPN formalism, and demonstrates how object-orientation helps to address a number of typical protocol modelling issues. The emphasis of the paper is practical – the theoretical foundations for this work are referenced in other documents. A few of the results presented here have been published elsewhere, but here they are collected together and presented in a uniform way in the simpler and more powerful notation of LOOPN++.

The textual language LOOPN++ which has been developed in tandem with the OPN formalism is presented in §2. The support for modularity and its relevance to the modelling of network protocols is addressed in §3, while §4 considers inheritance and §5 considers the aspect of genericity. The more advanced notion of supporting mobile objects is addressed briefly in §6. The concluding remarks are found in §7.

2 LOOPN++: a textual language for Object Petri Nets

A textual representation of Object Petri Nets, called LOOPN++, has been developed and implemented. As its name suggests, it is derived from its predecessor LOOPN [18], but is a significant advance in its incorporation of object-oriented facilities. LOOPN++ can serve as an object-oriented language in its own right, as a graphics-independent interchange format for Object Petri Nets, and as a temporary test bed while sophisticated graphical tools are being developed. The design of the language attempts to provide a *minimal set of contructs with orthogonal combinations*.

In this section, the LOOPN++ language is defined. The graphical conventions for Object Petri Nets are introduced with the examples in subsequent sections. The syntax is given in fig 2.1 and should

be read in conjunction with the text which follows. A fairly standard syntactic notation is used, together with the following metasymbols:

`[…]`	the enclosed construct(s) is optional
`{…}`	the enclosed construct(s) are repeated zero or more times with appropriate separators.

```
class   →  CLASS id [ : { parent } ]          -- parents + identifier renaming
              EXPORT  { ident } ;             -- list of externally accessible identifiers
              { field }                       -- fields hold data
              { func }                        -- functions evaluate expressions
              { action }                      -- actions change the data
              { trans }                       -- transitions consist of actions
           END id

field   →  type { ident [ = value ] } ;       -- fields with optional initial values

func    →  type  ident ( parms ) = expr ;     -- function with parameters and result expression

action  →  type  ident <- place  [ | expr ]   -- input action with selection condition
        →  type  ident -> place  [ | expr ]   -- output action with output value
        →  type  ident -- place  [ | expr ]   -- test action with selection condition
        →  proc-call                          -- interact with environment

trans   →  TRANS ident { action } END         -- transitions consist of a set of actions

place   →  ident

expr    →  value
        →  values-combined-with-operators     -- operators from underlying language

value   →  const
        →  ident
        →  '[' { ident:value } ']'            -- contructor for new object with fields
        →  ident '[' { ident:value } ']'      -- constructor for modified object
        → '[' { value } ']'                   -- multiset value constructor
        →  ident ( { expr } )                 -- value computed by function call

type    →  basic-type-ident                   -- int, bool, real, string, etc.
        →  class-ident                        -- predefined or user-defined
        →  type '*'                           -- multiset type for places
```

Fig 2.1: The basic grammar of LOOPN++

A LOOPN++ **program** consists of a finite set of **class** definitions, which are akin to the subnets of other formalisms. One class is designated the **root class** and a single instantiation of this class constitutes the main program or net. A **class** defines a set of **objects**, the **instances** of the class. The term **type** is used interchangeably with *class*. Types may be **predefined** (such as *boolean*, *integer*, *real*, *string*) or **user-defined**. A type of the form *type** indicates a multiset (or bag) of objects each of type *type*, and is referred to as a **multiset type**.

A **class** consists of fields, functions, actions and transitions. A **field** is a class component of some type which normally holds data (and hence determines part of the state of the object). Fields are allocated or bound on allocation of the enclosing object, but their contents may vary during their lifetime (especially if they perform the role of a Petri Net place).

A **function** defines a parameterised expression, which computes a value based on the other features (and hence state) of an object. The types of both parameters and function results may be simple, predefined types, or user-defined classes. Functions can use quantifiers: *forall*, *exists*, *count* to determine a value from the multiset (or bag) of tokens resident in a place.

An **action** may be an input action, an output action, a test action, or an anonymous action. An **anonymous action** consists of procedure calls which interact with the environment of the Petri Net. In order to support formal analysis it is desirable for anonymous actions to have no effect on the firing of transitions in the Petri Net. The other actions correspond to Petri Net arcs, and are the only constructs for changing the state of an object. The actions which are immediate components of an object are always synchronised with each other (like the arcs of a transition), but not necessarily with the actions contained within other component objects. Thus a transition is an instance of a class consisting only of actions (together with the appropriate binding of the incident places).

The **input**, **output**, and **test actions** all have a similar format:

```
type  x <- p | condition;        -- input action
type  x -> p | condition;        -- output action
type  x -- p | condition;        -- test action
```

Each specifies **token**(s) *x*, the associated **place** *p*, and an optional *condition*. For an input action to occur, the token(s) must be resident in the place, must be of appropriate type, and must satisfy the boolean *condition*, if any. The action removes the tokens from the place. For an output action to occur, the token(s) must be acceptable to the place, and must satisfy the boolean *condition*, if any. The action adds the tokens to the place. For a test action to occur, the token(s) must be resident in the place, must be of appropriate type, must *not* be selected for any input action, and must satisfy the boolean *condition*, if any. The action does not modify the tokens in the place.

For input and test actions, the *condition* may contain terms of the form:

```
x = value
```

which tests that the components of *x* match those specified by the value. For output actions, these should be the *only* terms contained in the *condition*, in which case they serve to generate objects with components matching those of the specified values.

It is important to note that output tokens are always newly-generated objects or newly-generated copies of existing objects. This ensures that tokens are always self-contained, and hence simplifies memory allocation and deallocation.

As indicated in the grammar, values may occur in a number of contexts – as the initial value of a field, as an expression computed by a function, or as a value associated with a token. The grammar indicates that such a value may be: a literal, a copy of an existing object, a new object with the values of individual fields specified, a copy of an existing object with certain fields modified, a list of values (for a multiset type), or a value computed from a function call.

3 Support for modularity

It has long been recognised [9, 27] that the successful management of complexity requires support for abstraction with clean module boundaries. This section considers some of the provisions of OPNs and LOOPN++ for building systems with clean module boundaries.

3.1 A protocol as a collection of services

OSI terminology [28] defines protocols as a collection of services and thus the networking standards documents tend to reflect this terminology and this modularisation of protocols. For example, the ANSI Z39.50-1992 Standard for Information Retrieval [1] speaks of a number of facilities – a Search Facility (for querying databases at a target), a Retrieval Facility (for retrieving copies of database records), a Result-set-delete Facility (for deleting result sets known to the target), an Access Control Facility (for allowing a target system to challenge an origin system), etc.

Unfortunately, the terminology and the concepts are not always consistently applied. Thus the Z39.50 standard identifies the distinct facilities as above and then presents the protocol as the monolithic transition table of fig 3.1. This makes it difficult to identify the states which belong to each service, and to separate the normal and abnormal activity of each service. This can only be done by reference to the informal, natural language description of the services. One suspects that this syle of specification encourages a lack of modularity.

It is clearly desirable for the OSI terminology of services to match the protocol definition. Even when this is not done, it is desirable to identify and model each service or facility separately in order to achieve an intellectually manageable model or specification. This has been done for the Z39.50 standard [23], from which we highlight a number of aspects below.

We commence by considering a number of definitions which are commonly required in the protocol description. The constant definitions for the various protocol data units (PDUs) are given in fig 3.2. Note that the graphical representation for a class is a labelled frame, with the components of the class drawn within the frame. Following Petri Net conventions for places, data fields are drawn as

ovals. They are labelled by the type, an identifier, and optionally an initial value. Note that a Petri Net place is simply a data field having a multiset type, i.e. one of the form *type**.

Fig 3.3 gives an excerpt from the class definition for PDUs. This is considered in further detail in §4. A similar form of definition will be used for the messages (MSGs) exchanged with the Z39.50 origin agent. Note that functions (like Petri Net transitions) are represented graphically as rectangles. Note also that the export of an item is indicated by drawing it with a double outline.

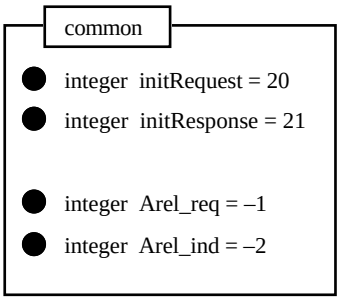
Table 10a: State Table for Origin – Part 1

State / Event	closed 1	Init sent 2	Open 3	Search sent 4	Prsnt sent 5	Delete sent 6	Rsrp Sent 7
Init req	Init PDU (2)						
Init resp PDU (ACCEPT)		Init conf + (30)					
Init resp PDU (REJECT)		Init conf –; Arel req (10)					
Srch req			Srch PDU (4)				
Srch resp PDU				Srch conf (3)			
Prsnt req			Prsnt PDU (5)				
Prsnt resp PDU					Prsnt conf (3)		
Dlte req			Dlte PDU (6)				
Dlte resp PDU						Dlte conf (3)	
Rsrp req			Rsrp PDU (7)				
Rsrp resp PDU							Rsrp conf (3)
Trigrc req		Trigrc PDU (2)		Trigrc PDU (4)	Trigrc PDU (5)	Trigrc PDU (6)	

Table 10a: State Table for Origin – Part 2

State / Event	Init sent 2	Open 3	Search sent 4	Prsnt sent 5	Delete sent 6	Rsrp Sent 7	Rsctrl recvd 8	Acctrl recvd 9	Rlease sent 10
Rsc PDU (Resp)	Rsc ind; stkst (8)		Rsc ind; stkst (8)	Rsc ind; stkst (8)	Rsc ind; stkst (8)	Rsc ind; stkst (8)			
Rsc PDU (Noresp)	Rsc ind (2)		Rsc ind (4)	Rsc ind (5)	Rsc ind (6)	Rsc ind (7)			
Rsc resp							Rsc resp PDU; popst		
Acc PDU	Acc ind; stkst (9)		Acc ind; stkst (9)	Acc ind; stkst (9)	Acc ind; stkst (9)	Acc ind; stkst (9)			
Acc resp								Acc resp PDU; popst	
Aab ind	Iab ind (1)	Iab ind (1)	Iab ind (1)	Iab ind (1)	Iab ind (1)	Iab ind (1)	Iab ind (1)	Iab ind (1)	Iab ind (1)
Apab ind	Iab ind (1)	Iab ind (1)	Iab ind (1)	Iab ind (1)	Iab ind (1)	Iab ind (1)	Iab ind (1)	Iab ind (1)	Iab ind (1)
Iab req	Aab req (1)	Aab req (1)	Aab req (1)	Aab req (1)	Aab req (1)	Aab req (1)	Aab req (1)	Aab req (1)	Aab req (1)
Irel req		Arel req (10)							
Arel conf									Irel conf (1)

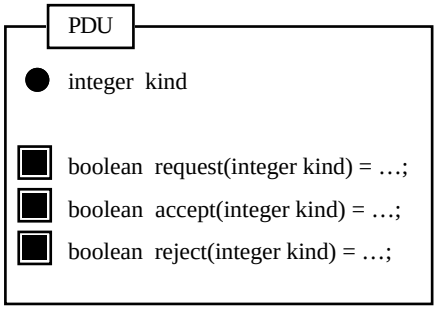
Fig 3.1: Transition table for Z39.50




```
Class common
  // Z39.50 PDU kinds
  integer  initRequest = 20;
  integer  initResponse = 21;
  …
  // Association service PDU kinds
  integer  Arel_req = -1;
  integer  Arel_ind = -2;
  …
End common
```

Fig 3.2: Graphical and textual representation of the class with common constants

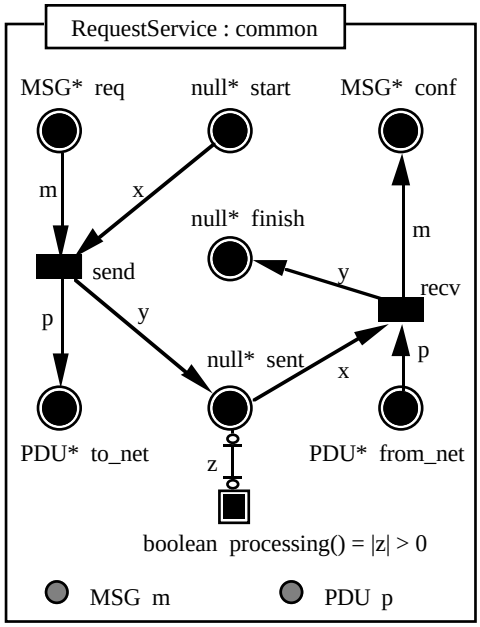
In modelling the various protocol services, it will be necessary to test whether a PDU is a request, an accepting response, or a rejecting response for that particular service. A natural object oriented solution is to encapsulate the appropriate boolean functions with the class definitions for PDUs (and MSGs), as shown in fig 3.3.




```
Class PDU
  Export request, accept, reject;
  // PDU data fields
  integer  kind;
  …
  // PDU functions
  boolean request(integer kind) = …;
  boolean accept(integer kind) = …;
  boolean reject(integer kind) = …;
  …
End PDU
```

Fig 3.3: Graphical and textual representation of the class for Protocol Data Units (PDUs)

Careful study of the Z39.50 standard reveals that a number of the origin-initiated services (including Initialize, Release, Search, Present, Delete, Resource-report) can be captured by a particular style of subnet with an initial, intermediate, and final state (as shown in fig 3.4). This class can then be instantiated for each of the above services, possibly fusing the initial and final states.

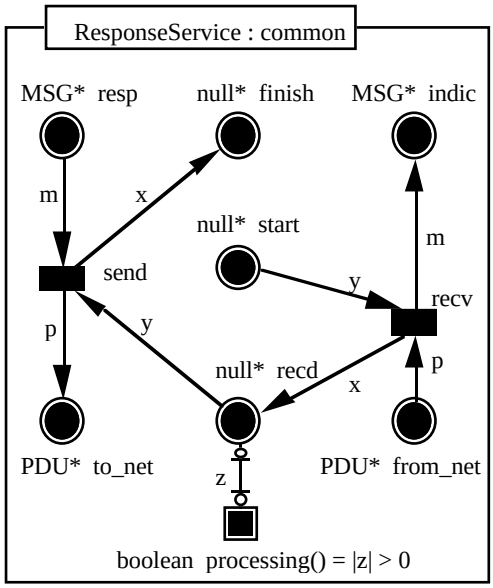



```
Class RequestService : common
  Export service, req, conf, to_net, from_net,
         start, sent, finish, processing;
  integer service;
  null* start, sent, finish;
  MSG*  req, conf;
  PDU*  to_net, from_net;
  boolean processing() = exists z <- sent;
  trans send
     null x <- start;
     MSG  m <- req | m.request(service);
     null y -> processing;
     PDU  p -> to_net | …;
  end
  trans recv
     null x <- processing;
     PDU  p <- from_net | p.accept(service);
     null y -> finish
     MSG  m -> conf | …;
  end
End RequestService
```

Fig 3.4: Graphical and textual representation of a request service

Note that places are identified by their multiset type. In this class, all the places are exported and hence may be bound to external places when this class is instantiated. (This binding persists for the life of the class instance.) The predefined type *null* is a token type with no data fields, but with some predefined functions. The arcs are annotated with token variables, which are represented graphically as shown for *m* and *p*. (*x* and *y* have not been shown in the interests of brevity.) The striped shading indicates that the value is bound to the variable for the duration of transition firing. The function *processing* is defined to return *true* if the place *sent* contains at least one token.

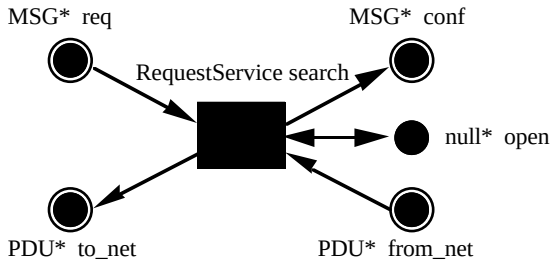
Following the conventions of [17], this is shown graphically with the use of an equal arc, i.e. an arc which is enabled only if its inscription (here the variable *z*) is identical to the marking of the place (here *sent*). A similar class is defined for target-initiated services, as in fig 3.5.




```
Class ResponseService : common
  Export service, resp, indic, to_net, from_net,
         start, recd, finish, sending;
  integer service;
  null* start, sent, finish;
  MSG*  req, conf;
  PDU*  to_net, from_net;
  boolean processing() = exists z <- recd;
  trans recv
     null y <- start;
     PDU  p <- from_net | p.request(service);
     null x -> recd
     MSG  m -> indic | …;
  end
  trans send
     null y <- processing;
     MSG  m <- req | m.accept(service);
     null x -> finish;
     PDU  p -> to_net | …;
  end
End ResponseService
```

Fig 3.5: Graphical and textual representation of a response service

The Z39.50 Search facility, like a number of other facilities, can be modelled by instantiating the request service as in fig 3.6. Since this is just one component of the Z39.50 origin entity, it has been drawn without a class boundary. The textual annotation (which should all appear on the diagram) indicates that *search* is an instance of the *RequestService* class, that the exported field *service* is bound to the constant *searchRequest*, that the exported places *req*, *conf*, *to_net*, *from_net* are bound to similarly-named places in the context of the instance, that the exported places *start* and *finish* are both bound to the place *open* (which holds a token once a connection has been established), and that the exported place *sent* is not bound, and hence will be local to the instance.




```
RequestService search =
          [service:searchRequest,
           req:req, conf:conf,
           to_net:to_net, from:net:from_net,
           start:open, finish:open];
```

Fig 3.6: Graphical and textual representation of the instantiation of the Search request service

This instance, together with similar instances for the Present, Delete and Resource-report facilities, will cover the entries in part 1 of the state transition table (of fig 3.1) for states 3, 4, 5, 6, 7. The same class is instantiated for the Initialize and Release services (states 1, 2, 10), but their interaction needs to be captured, and hence we define a combined service, as in fig 3.7.

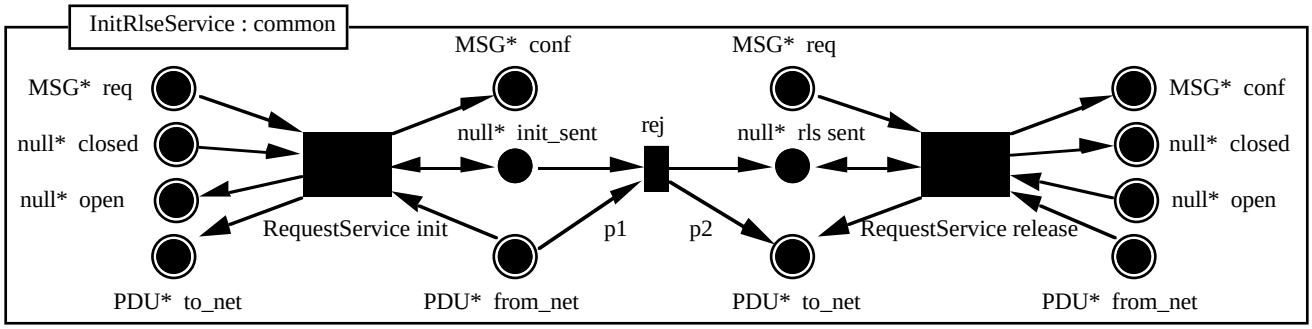


Fig 3.7: Graphical representation of the Initialze and Release services

Note that the various places have been duplicated or aliased to avoid crossing arcs. Note also that the Initialize facility normally takes you from the *closed* to the *open* state, via the *init_sent* state, while the Release facility normally takes you from the *open* to the *closed* state, via the *rls_sent* state. However, an Initialize request may be rejected, in which case a Release request is sent. This is achieved by the transition *rej* which would have a guard of the form *p1.reject(initRequest)*.

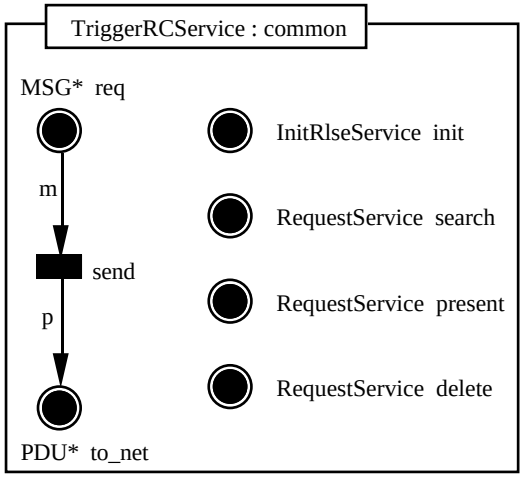
The above demonstrates how it is convenient to capture the various services of the Z39.50 protocol as subnets. Such modularity is readily supported by many Petri Net formalisms such as CPNs [13]. In the following sections, further aspects of the protocol are discussed using OPN facilities which are not readily supported by other Petri Net formalisms.

3.2 Weak coupling of subnets

As already noted, the modular construction of a net by instantiating a number of subnets interacting with shared places is a common form of net decomposition. It is often the only one. For example, the original proposal for Hierarchical CPNs [10] advocated place fusion, transition fusion and invocation transitions. Only the first has been implemented in the commercially available package Design/CPN [14] while the second is common in other object-oriented net formalisms [4, 7]. This implies that the only way to interact with a subnet is to exchange tokens with it or to synchronise with one of its transitions. The logic of the subnet must then cater for every distinct interaction style. Even if the interaction simply involves examining some aspect of the state, the subnet needs to explicitly receive the request and return the result. This is contrary to the principle enunciated by Meyer [24] that objects should have high internal cohesion and weak external coupling.

A fundamental technique for achieving weak coupling between objects in object-oriented languages is to define exported functions which can evaluate selected aspects of an object's state without changing that state. It seems that Petri Net models have generally been slow to adopt this fundamental technique. LOOPN++ (like its predecessor LOOPN) supports the definition of access functions which may examine the state of a subnet. Elsewhere [17] it has been shown that this provision can be formally defined so as to support the usual step semantics of Petri Nets, and furthermore, nets with such extensions can be mapped into behaviourally equivalent CPNs.

In modelling the Z39.50 protocol, the advantages of this facility in maintaining modularity can be demonstrated by considering the Trigger-resource-control Service and the Resource-control Service (where no response is required). Both of these require the origin entity to have sent an Initialize, Search, Present or Delete request, but not to have received a corresponding reply. In other words, they require one of the listed services to be in their intermediate state. While the place indicating this intermediate state has been declared as exported (fig 3.4), this has only been exploited in the Initialize-Release service (fig 3.7). This encapsulation should be maintained in the interests of the weak coupling of subnets. Accordingly, a function *processing* was defined for the various services to determine whether the subnet was in its intermediate state. This now makes it possible to define the Trigger-resource-control service as in fig 3.8.




```
Class TriggerRCService : common
  Export req, to_net,
         init, search, present, delete;
  integer service = triggerResourceControlRequest;
  MSG*  req;
  PDU*  to_net;
  InitRlseService init;
  RequestService  search, present, delete;
  trans send
     MSG  m <- req | m.request(service) and
                     (init.processing() or
                      search.processing() or
                      present.processing() or
                      delete.processing());
     PDU  p -> to_net | …;
  end
End TriggerRCService
```

Fig 3.8: Graphical and textual representation of the Trigger-resource-control service

Note that this subnet assumes that the various services will be bound to the local data fields so that they can be used to interrogate the state of those services.

The advantages of supporting subnet access functions has also been demonstrated in the modelling of an electronic door controller protocol at the University of Tasmania [22]. Here, each door in a building was equipped with a microprocessor which could sense and modify the state of the door, manage the associated keypad and display unit, and communicate with the central PC controller. The application layer protocol was modelled in LOOPN in order to isolate potential protocol failure and possible timing problems.

Where possible, interaction between subnets was achieved with the use of access functions, thus minimising the amount of token-passing. This facilitated the flexible combination of subnets. For example, a fire door does not have a keypad and display unit, and it turned out to be a simple matter to assemble an application layer protocol for these doors which simply omitted instantiation of the keypad and display subnets.

It needs to be emphasised that this kind of flexibility is possible because of the weak coupling achieved by the use of access functions, which allow the state of a subnet to be examined without modifying that state. What is true of this simple example is even more imperative for complex systems. As far as possible, the complexity of module interactions should be minimised.

4 Support for inheritance and polymorphism

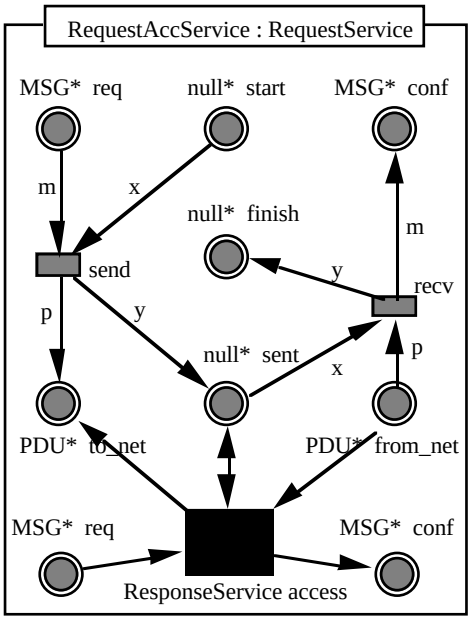
Languages which support the definition and instantiation of modules are classified as *object-based* [33]. Such languages encourage a certain amount of software reuse. Thus, in §3.1 the Z39.50 protocol was built up as a number of services, each of which was the instantiation of one of two classes.

One of the primary motivations for the development of object-oriented technology was the quest for more effective mechanisms for software reuse [24]. This has primarily been achieved through inheritance and polymorphism, which qualify a language to be described as *object-oriented* [33]. Inheritance allows a class to derive its features from another (its parent class), and then to augment them or modify them. Then, polymorphism means that an instance of the subclass may be used in a context specifying the parent. This facility has made possible the explosive growth in application frameworks, particularly in the realm of graphical user interfaces [2, 32, 34].

OPNs and LOOPN++ support inheritance and polymorphism, and can thus benefit from this style of software reuse. One aspect of this can be demonstrated in the case of the Z39.50 protocol, by considering the definition of the classes *RequestService* and *ResponseService* (figs 3.4 and 3.5). The logic of these subnets depends on receiving and sending tokens, which have functions defined to test whether the token is a request or a response for a particular service. The same logic applies irrespective of any additional information carried by the token. Thus, in modelling the details of the Z39.50 protocol, it is convenient to define primitive *PDU* and *MSG* classes as in fig 3.3, and then to define specific PDU formats as subclasses of these. The logic of the above subnets can be used unchanged irrespective of the particular PDU format required for each service.

A further demonstration of software reuse enabled by inheritance and polymorphism can be made by considering the provision of access control and error-handling in the Z39.50 protocol. These were not examined in §3 where attention was focussed on what might be called the normal operation of each service. It will be noted from part 2 of the transition table of fig 3.1 that Access control temporarily interrupts a service (with a transfer to state 9). The handling of abort requests permanently interrupts a service. It is possible to include these extensions for each facility as net components undifferentiated from the normal operation, but this can all too easily obscure the central function of the service (or facility). With OPNs, it is possible (and desirable) to specify these extensions as extensions of the normal service. Thus, the generic *RequestService* of fig 3.4 can be extended to a generic service with access control as shown in fig 4.1. This *RequestAccService* inherits from *RequestService*, with inherited components shown graphically by a grey shading. *RequestService* is augmented with an instance of *ResponseService*, which responds to the Access Control request.

A similar approach can be taken to building service components with abort functionality, except that all such service components will need to have access to the *closed* state. The Z39.50 origin entity can then be defined as a collection of these extended services.




```
Class RequestAccService : RequestService
  ResponseService access =
                  [service:accessControlRequest,
                   req:req,
                   conf:conf,
                   to_net:to_net,
                   from_net:from_net,
                   start:sent,
                   finish:sent];
End RequestAccService
```

Fig 4.1: Graphical and textual version of a request service with access control

The above examples may be considered as reuse of software during design. We have also found that software reuse happens in practice across projects. A simple example of this reuse occurred in the development of the lower layer protocols for the door controller discussed briefly in §3.2. Prior to starting that project, we had developed a Data Link Layer protocol for point-to-point communication. In modelling the door controller, it was necessary to have a Data Link Layer protocol for a multidrop environment. It was a relatively simple matter to modify the point-to-point version by transmitting tokens containing the door number (rather than *null* tokens), and by modifying the reception of tokens so that only tokens destined for the current door were accepted.

5 Support for genericity

One of the primary motivations for the development of the OSI Reference Model [11] was the appropriate modularisation of the complexities associated with networking. Each layer of the Model addressed certain issues and delegated others to lower layers. A consequence of this is that a layer should not be concerned with higher layer issues, but simply provide a set of services to these higher layers through a number of primitives. As a result, any data submitted for transmission by a higher layer via one of these primitives should remain uninterpreted. Thus a network layer receiving a packet for transmission to some destination node should perform in a similar way independent of the particular application requesting the transfer.

In other words, the layering of protocols demands the ability to define generic software components. One simple approach (and the one commonly implemented in networking software) is to treat the data simply as a sequence of bits or bytes (octets) and to supply appropriate encoding and decoding routines. In the abstract modelling of protocols, it is preferable to retain the type information, in which case the protocol layers will need to be able to transfer a variety of differently typed messages, and hence the need for genericity.

In modelling such generic protocol layers, OPNs and LOOPN++ can take advantage of the support for polymorphism already discussed in §4. A protocol layer defined to transfer tokens of a given class, can also be used to transfer tokens of any subclass. The simplest case in LOOPN++ is to define a protocol layer to transfer tokens of type *null*, which is a predefined class having built-in functions but no data. Since every LOOPN++ class inherits from *null*, the protocol layer will be able to transfer any token type.

There is a subtle twist to this strategy, since a protocol layer will normally add some header information on transmission, and remove it on reception. For example, a data link layer will typically add a header including the kind of frame, its sequence number and its acknowledgement number. Then the kind of token handled by the higher layer interface will differ from the kind of token handled by the lower layer interface. It is then important that the proposed polymorphic use of the layer will be consistently defined. This issue can be highlighted by examining typical transitions which would add and delete such header information, as shown in fig 5.1.

```
TRANS send
    null x <- network_layer | can_accept_message;
    seq  y -> physical_layer | y = x [seq: nextseq(), …];
END
TRANS recv
    seq  x <- physical_layer | sequence_number_is_OK;
    null y -> network layer | y = x;
END
```

Fig 5.1: Generic send and receive transitions

Note that the *send* transition adds a sequence number to the incoming token, while the transition *recv* removes it. We assume the following notation:

(a) *class(x), class(y)* is the declared class of (tokens) *x* and *y*
(b) *class(x)* $\leq$*: class(y)* means that the declared class of *x* is a subclass of that of *y*
(c) given *class(x)* $\leq$*: class(y)*, we write *class(x) = class(y) + C*, where *C* is the class that augments *class(y)* to give *class(x)*
(d) *class(x'), class(y')* is the actual class of the tokens bound to *x* and *y* at run-time

In order to use the *send* and *recv* transitions generically, the following conditions must hold:

(e) *class(x') = class(x) + C* $\leq$*: class(x)*
(f) *class(y') = class(y) + C* $\leq$*: class(y)*

Both points (e) and (f) state that the actual token class is a subclass of the declared class. They also demand that both augment the declared class in the same way. In other words, the hidden information, given by the class *C* is transferred intact.

The above constraints can be (and have been) implemented with run-time type-checking. It is also desirable to be able to guarantee type safety at compile time. In order to support this, languages like Eiffel and C++ [25, 30] define generic classes with a type parameter which is then specified each time the class is instantiated. We prefer the more flexible approach of Palsberg and Schwartzbach [26], which allows *any* component class to be consistently renamed while still retaining type safety.

6 Support for mobile objects

As network protocols become increasingly complex, it is likely that the messages transferred will encapsulate more highly structured information, including objects which encapsulate both data and operations. For example, mail messages may well encapsulate functionality to selectively display parts of the message depending on the security clearance of the reader, or may vary the display depending on the display terminal available. In the world of Electronic Data Interchange [15], documents are already highly structured objects, possibly containing their own paper trail, in the form of subdocuments.

Such increasingly complex messages demand more powerful structuring facilities, if appropriate modelling is to be possible. In §3.1 the Z39.50 protocol data units were modelled as classes encapsulating both data fields and functions. Elsewhere, we have shown how the notion of mobile objects suitable for EDI documents can be conveniently modelled in OPNs [20].

7 Conclusions

This paper has presented LOOPN++, a textual language for the formalism of Object Petri Nets. There is an economy of notions in LOOPN++. For example, types may encompass simple predefined types, user-defined classes without actions, and user-defined classes with actions. Hence the notion

of a field encompasses a number of different notions from more traditional petri net formalisms – simple constants, Petri Net places, and subnet instances. The same flexible type system applied to functions and tokens means that tokens may be associated with subnets and functions may return subnet instances. (Just as the form *type** is used for multiset types, so the extended forms *type***, *type****, etc. are allowable, even if not commonly used!) The uniform type system therefore caters directly for the arbitrary nesting of objects, which sets OPNs and LOOPN++ apart from other object-oriented net formalisms [3, 4, 7, 29, 31].

The grammar for LOOPN++ does not specify the form of expressions. As with Coloured Petri Nets [13] it is possible to have a number of flavours of LOOPN++ depending on the underlying expression syntax and semantics – from the richer functional programming style to the more constrained C++ style (as currently implemented). As with CPNs, it is sufficient for the formalism of Object Petri Nets to be able to determine the type of an expression and to be able to evaluate it in some context.

A class may be declared to inherit the features of one or more parents, in which case all the features of the parents, together with the additional features declared within the class constitute the features of the new class. It is possible to override one feature by another provided it is of the same kind (i.e. field, function or action) – it is not possible to override a field by a function or an action.

Petri Net transitions are supported as syntactic sugar, not as fundamental constructs. Any class containing actions can change the state of the object and thus has transition properties. In the same way, any object which can accept or offer tokens to its environment can act in the role of a place. This makes possible the notion of a superplace, where the acceptance or offer of tokens to the environment is controlled by the internal logic of the object. It is thus a trivial matter to define a capacity place in LOOPN++. The offer of token(s) by a superplace to its environment is synchronised with an action of the environment in removing the tokens from the object.

In summary, OPNs and LOOPN++ incorporate a uniform and flexible type system which provides good support for modularity, inheritance, polymorphism, genericity, and mobile objects. This paper has demonstrated how each of these attributes is of significant benefit when modelling network protocols.

While this paper has not presented the formal foundations for OPNs, other papers have been referenced which give the formal definition and prove that OPNs can be transformed into behaviourally-equivalent CPNs. This is a first step towards adapting analysis techniques developed for CPNs to OPNs. Consequently, OPNs retain the attributes of formal definition, executable models, and amenability to automated analysis, which have facilitated earlier application of Petri Nets to the modelling and analysis of network protocols.

Currently, a preliminary version of LOOPN++ has been implemented. By translating LOOPN++ programs into C++, it is intended that it will be easy to integrate LOOPN++ with other software packages, either to provide analysis tools, or to provide a prototyping environment. For example, in developing the door controller protocol of §3.2, it was observed that such an open environment can lead to a significant part of a protocol model being reused as the foundation of a prototype implementation. As a further experiment in providing an open environment, the integration of LOOPN++ with a persistent object store is currently being investigated.

It is therefore anticipated that OPNs will reap the practical benefits of object-orientation including clean interfaces, reusable software components, and extensible component libraries.

Acknowledgements The authors gratefully acknowledge the contributions of the reviewers, which have served to improve the quality of this paper.

References

[1] ANSI *Z39.50: Information Retrieval Service and Protocol* ANSI/NISO Z39.50-1992 (version 2), American National Standards Institute (1992).

[2] Apple Corporation *MacApp Reference Manual* (1992).

[3] M. Baldassari and G. Bruno *An Environment for Object-Oriented Conceptual Programming Based on PROT Nets* Advances in Petri Nets 1988, G. Rozenberg (ed.), Lecture Notes in Computer Science 340, pp 1–19,

Springer Verlag (1988).
[4] E. Battiston, F. de Cindio, and G. Mauri *OBJSA Nets: A Class of High-level Nets having Objects as Domains* Advances in Petri Nets 1988, G. Rozenberg (ed.), Lecture Notes in Computer Science 340, pp 20–43, Springer-Verlag (1988).
[5] G. Berthelot and R. Terrat *Petri Nets Theory for the Correctness of Protocols* Proceedings of Protocol Specification, Testing and Verification II, pp 325-341, Los Angeles, North-Holland (1982).
[6] J. Billington, G.R. Wheeler, and M.C. Wilbur-Ham *PROTEAN: A High-Level Petri Net Tool for the Specification and Verification of Communication Protocols.* IEEE Transactions on Software Engineering, **14**, 3, pp 301–316 (1988).
[7] D. Buchs and N. Guelfi *CO-OPN: A Concurrent Object Oriented Petri Net Approach* Proceedings of 12th International Conference on the Application and Theory of Petri Nets, Gjern, Denmark (1991).
[8] M. Diaz *Modelling and Analysis of Communication and Cooperation Protocols Using Petri Net Based Models.* Proceedings of Protocol Specification, Testing and Verification II, pp 419–441, Los Angeles, North-Holland (1982).
[9] E.W. Dijkstra *Notes on Structured Programming* Structured Programming, O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare (eds.), pp 1-82, Academic Press (1972).
[10] P. Huber, K. Jensen, and R.M. Shapiro *Hierarchies of Coloured Petri Nets* Proceedings of 10th International Conference on Application and Theory of Petri Nets, Lecture Notes in Computer Science 483, pp 313-341, Springer-Verlag (1990).
[11] ISO *Information Processing Systems – Open Systems Interconnection: Basic Reference Model* International Organisation for Standardization and International Electrotechnical Committee (1984).
[12] K. Jensen *Coloured Petri Nets: A High Level Language for System Design and Analysis* Advances in Petri Nets 1990, G. Rozenberg (ed.), Lecture Notes in Computer Science 483, Springer-Verlag (1990).
[13] K. Jensen *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use – Volume 1: Basic Concepts* EATCS Monographs in Computer Science, Vol. 26, Springer-Verlag (1992).
[14] K. Jensen, S. Christensen, P. Huber, and M. Holla *Design/CPN™: A Reference Manual* MetaSoftware Corporation (1992).
[15] P. Kimberley *Electronic Data Interchange* McGraw-Hill (1991).
[16] R. Lai, T.S. Dillon, and K.R. Parker *Verification Results for ISO FTAM Basic Protocol* Proceedings of Protocol Specification, Testing and Verification IX, pp 223-234, North-Holland (1990).
[17] C. Lakos and S. Christensen *A General Systematic Approach to Arc Extensions for Coloured Petri Nets* Proceedings of 15th International Conference on the Application and Theory of Petri Nets, Lecture Notes in Computer Science 815, pp 338-357, Zaragoza, Springer-Verlag (1994).
[18] C.A. Lakos *LOOPN User Manual* Technical Report , Department of Computer Science, University of Tasmania (1992).
[19] C.A. Lakos *Object Petri Nets – Definition and Relationship to Coloured Nets* Technical Report TR94-3, Computer Science Department, University of Tasmania (1994).
[20] C.A. Lakos *From Coloured Petri Nets to Object Petri Nets* Proceedings of 15th International Conference on the Application and Theory of Petri Nets, Lecture Notes in Computer Science (*To appear*), Torino, Springer-Verlag (1995).
[21] C.A. Lakos and C.D. Keen *Modelling Layered Protocols in LOOPN* Proceedings of Fourth International Workshop on Petri Nets and Performance Models, Melbourne, Australia (1991).
[22] C.A. Lakos and C.D. Keen *Modelling a Door Controller Protocol in LOOPN* Proceedings of 10th European Conference on the Technology of Object-oriented Languages and Systems, Versailles, Prentice-Hall (1993).
[23] J.W. Lamp *Encoding the ANSI Z39.50 Search and Retrieval Protocol using LOOPN* Honours Thesis, Department of Computer Science, University of Tasmania (1994).
[24] B. Meyer *Object-Oriented Software Construction* Prentice Hall (1988).
[25] B. Meyer *Eiffel: The Language* Prentice Hall (1992).
[26] J. Palsberg and M.I. Schwartzbach *Object-Oriented Type Systems* Wiley Professional Computing, Wiley (1994).
[27] D.L. Parnas *On the Criteria to be Used in Decomposing Systems into Modules* CACM, **15**, 12, pp 1053-1058 (1972).
[28] M.T. Rose *The Open Book: A Practical Perspective on OSI* Prentice-Hall (1990).
[29] C. Sibertin-Blanc *Cooperative Nets* Proceedings of 15th International Conference on the Application and Theory of Petri Nets, Lecture Notes in Computer Science 815, pp 471-490, Zaragoza, Spain, Springer-Verlag (1994).
[30] B. Stroustrup *The C++ Programming Language (Second Edition)* Addison-Wesley (1991).
[31] P.A.C. Verkoulen *Integrated Information Systems Design: An Approach Based on Object-Oriented Concepts and Petri Nets* PhD Thesis, Technical University of Eindhoven, the Netherlands (1993).
[32] J.M. Vlissides *Generalized Graphical Object Editing* Technical Report CSL-TR-90-427, Stanford University (1990).
[33] P. Wegner *Dimensions of Object-Based Language Design* Proceedings of OOPSLA 87, pp 168-182, Orlando, Florida, ACM (1987).
[34] A. Weinand, E. Gamma, and R. Marty *ET++ – An Object-Oriented Application Framework in C++* Proceedings of OOPSLA 88 Conference, ACM (1988).