The Incremental Modelling of the Z39.50 Protocol with Object Petri Nets

Charles Lakos[1] and John Lamp[2]
[1]Computer Science, University of Adelaide,
Adelaide, SA, 5005, Australia.
[2]Management Information Systems, Deakin University,
Waurn Ponds, Geelong, Victoria, 3217, Australia.
Charles.Lakos@cs.adelaide.edu.au
John.Lamp@deakin.edu.au

Abstract: This paper examines how object-oriented extensions to the Petri Net formalism provide flexible structuring primitives which can aid the modelling of network protocols. A key benefit is the support for incremental modelling. As a result, a protocol can be modelled as a collection of services, each of which can be expressed as the enhancement of a basic service, in which case, both the structure of the basic service and the nature of the enhancement can be clearly identified. More importantly, the evolution of a protocol through a sequence of standards can be expressed by progressive refinements. The object-oriented extensions are captured in the formalism of Object Petri Nets, with a textual language form referred to as LOOPN++, both of which are introduced in this paper. The incremental modelling capabilities and their benefits are demonstrated for the Z39.50 Protocol for Information Retrieval.

1 Introduction

For a long time, computer network protocols have motivated research in concurrent systems. The increasing complexity of concurrent systems 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. Currently, for example, there is much interest in client-server applications and distributed object computing.

One formalism which has been applied to network protocols with beneficial results is that of Petri Nets [7, 8, 12, 21]. This has been facilitated by a number of attributes traditionally associated with Petri Nets – their formal definition, their graphical representation, the associated executable models, and their 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* [16]. This is primarily seen in the limited facilities for building complex systems out of simple components. Equally important there is a lack of facilities for building systems incrementally.

Network protocols, like software systems in general, have a number of aspects which are particularly suited to incremental modelling. They are typically composed of a number of protocol services, where each service is best understood (and modelled) as a basic service together with enhancements. Network protocols tend to have a number of configurable options. Finally, network protocols (like software systems in general) tend to evolve over time. It is desirable to be able to capture these enhancements, configurations and evolution directly with incremental specifications, rather than having to start with a fresh definition or having a number of specifications.

This paper considers the modelling of the Z39.50 Protocol for Information Retrieval [1, 2] which exhibits all of the above properties. It consists of a number of services which are best captured as basic services together with enhancements, it has a number of configurable options, and the protocol has evolved recently from the 1992 to the 1995 version.

The intent of the Z39.50 standard is to provide the kernel of a client/server system which allows computer-to-computer information search and retrieval. Z39.50 does not prescribe the way information is managed at the server, nor does it prescribe how information is presented at the client. This allows for a wide range of information sources to be accessed using Z39.50, with a client whose capabilities may range from the simplistic, to an intelligent software agent embedded in a larger information system.

The Z39.50 standard was initially proposed by the librarianship community to provide an open standard for networked access to bibliographic databases, and has now been extended to handle other forms of textual and non-textual databases such as graphical and geographic databases. It has also been used to support searches based on generalised pattern-matching techniques. These techniques will become increasingly important in the applications currently being developed for finding abstract information such as chemical structures, gene sequences, fingerprints, faces, video imagery, and numeric trend data. There are hundreds of information sources using Z39.50 or Wide Area Information Server (WAIS) protocols. (WAIS was based on an early version of Z39.50.)

The significance of Z39.50 has been recognised by the US Federal Government who have specified its use as part of the Government Information Locator Service (GILS). A number of US government agencies use Z39.50, as well as other government agencies such as the European Space Agency, many national libraries and universities, and commercial information sources such as Dialog, LEXIS/NEXIS, Online Computer Library Center (OCLC) and the Research Libraries Group. There are both freeware and commercial clients and servers available, as well as gateways for resources such as X.500, SQL and HTTP.

This paper is an extended version of an earlier one [31] which restricted its attention to Z39.50-1992 since Z39.50-1995 had not been standardised at that stage. By modelling the 1995 version as an extension of the 1992 version, this paper exhibits a realistic case study of incremental modelling.

The Petri Net formalism we use is called Object Petri Nets (OPNs) [24, 27], which is a modified Coloured Petri Net (CPN) formalism [17] incorporating object-oriented structuring. 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 extensibility and software reuse. An implementation of this formalism is the textual language LOOPN++.

The paper introduces both the graphical conventions for OPNs and the textual form of LOOPN++ in §2. The emphasis of this presentation is practical – the theoretical foundations for this work are found elsewhere [24, 27]. Z39.50-1992 is introduced in §3, its basic services are modelled in §4 and the enhanced services in §5. Z39.50-1995 is then introduced in §6, its new services in §7 and other significant extensions in §8. A number of other issues pertinent to protocol modelling are noted in §9, while §10 introduces the issues which are pertinent to the analysis of OPNs. Finally, the conclusions and proposals for further work are presented in §11.

2 Introduction to Object Petri Nets and LOOPN++

In this section we introduce Object Petri Nets (OPNs), their graphical conventions and their textual representation in the language LOOPN++. Each subsection begins with the relevant segment of the grammar for LOOPN++, and the following text discusses the implications of the grammar together with the associated graphical conventions. The grammar captures the possibilities of the language in a simple and concise manner, and provides a minimal set of contructs with orthogonal combinations. The graphical conventions extend the traditional Petri Net notation and are complemented by the textual representation, which supplies the annotations for the diagrams. Many of these annotations are only selectively displayed, so as to avoid cluttering the diagrams. Thus, the textual form can serve as an object-oriented

language in its own right, as a graphics-independent interchange format for OPNs, and as a temporary test bed while sophisticated graphical tools are being developed. Currently, only the textual form has been implemented, but a graphical editor is nearing completion.

2.1 Nets, classes and instances

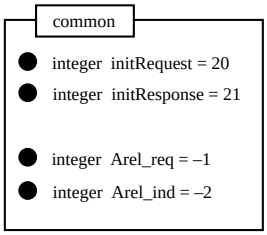
```
class   →  CLASS id [ : [ parent ] {, parent } ]   -- parents
             EXPORT ident {, ident }                -- exported identifiers
             { field }                              -- data fields
             { func }                               -- functions
             { trans }                              -- transitions
             { action }                             -- token + anonymous actions
           END id
```

An OPN specification or **net** consists of one or more **class** definitions. 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 user-defined classes or basic, predefined types such as `boolean`, `integer`, `real`, `char`, `string`.

In general, a class consists of a number of components, each of which is a **field** (or data), **function**, **transition** or **action**. These are allocated and initialised on instantiation of the enclosing object.

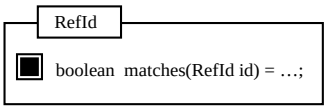
Graphically, a class is drawn as a labelled frame enclosing the class components, as in fig 2.1. The label on the frame specifies the class name and the parent classes. This example shows a class with a number of constant definitions which are used in the definition of the Z39.50 protocol.




```
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 2.1: Graphical and textual representation of the class with common constants

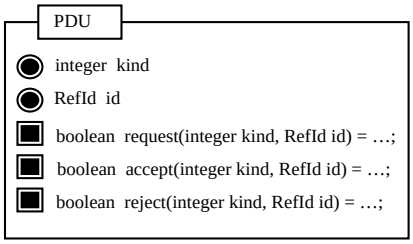
Following Petri Net conventions, state components are drawn as ovals (or circles), and state change (or computation) components are drawn as rectangles (or squares). Fig 2.2 shows a class with a function component. While a function does not change the state, it does represent a computation which is performed at a certain point in time, and hence is drawn as a rectangle.




```
Class RefId
  export matches;
  boolean matches(RefId id) = …;
End RefId
```

Fig 2.2: Graphical and textual representation of the class for the reference id

Classes may be arbitrarily instantiated as components or features of other classes. Thus, fig 2.3 shows a class with one field instantiating the class *RefId* (of fig 2.2).

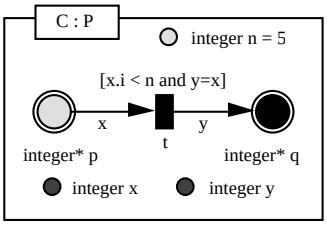



```
Class PDU
  Export kind, id, request, accept, reject;
  // PDU data fields
  integer  kind;
  RefId    id;
  …
  // PDU functions
  boolean request(integer kind, RefId id) = …;
  boolean accept(integer kind, RefId id) = …;
  boolean reject(integer kind, RefId id) = …;
  …
End PDU
```

Fig 2.3: Graphical and textual representation of the class for Protocol Data Units (PDUs)

Each feature of a class may be local or exported, with only exported features externally visible and accessible. Textually, an **export** clause lists the identifiers which are exported. Graphically, an exported component is shown with a double outline. Thus, figs 2.2 and 2.3 show classes where all components are exported, while fig 2.1 shows a class where no components are exported.

A class may **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 for a class to override a feature of the parent with a compatible one of the same kind – it is not possible to override a field by a function or an action. Graphically, inheritance is shown using two conventions – the parent classes are listed in the label of the class frame, and the components inherited without change are drawn with a grey shading, as in fig 2.4. In this example, class *C* inherits from class *P*. Class *C* has features *n*, *p*, *q*, *t* plus associated arcs. Components *n*, *p* are inherited without change, while *q*, *t* are introduced locally or override similarly named components of parent *P*. Only *p* and *q* are exported. Note that transition variables *x*, *y* can also be shown graphically, with a striped shading to emphasise their temporary nature.




```
CLASS C : P
 export q;
  integer* q;
  trans t
    integer x <- p | x.i < n;
    integer y -> q | y = x;
  end t
END C
```

Fig 2.4: Graphical and textual representation of a class C which inherits from parent class P

OPNs support the usual **polymorphism** of object-oriented systems, which allows instances of a subclass to occur in superclass contexts.

Both OPNs and LOOPN++ maintain the bipartite nature of Petri Nets by distinguishing between state classes and transition classes. A **transition class** is one with arcs as (immediate) components, while a class without such arc components is a **state class**. (A state class may contain transitions but may *not* contain arcs.)

2.2 Types

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

Various net components may have a **type** which may be any built-in, predefined type or a user-defined class. The type may also be of the form `type*` which indicates a multiset (or bag) of objects each of type `type`. (The form `type*` is used to declare Petri Net places.)

2.3 Values and expressions

```
value   →  ident                                          -- copy of specified object
        →  '['ident:value {, ident:value }']'             -- new object with specified fields
        →  ident'['ident:value {,ident:value }']'         -- modify object by specified fields
        →  '['value {, value} [ | value ]']'             -- multiset value constructor
        →  expr                                           -- value given by expression
```

In certain contexts – in field and function definitions and in guards – it is possible to specify values. These values may indicate a copy of an existing object, the generation of a new object (by specifying values for some of the exported fields), a copy of an existing object with certain specified fields modified, a multiset of values, or a value computed by an expression. For a multiset of values, it is possible to specify a tail or remainder (following the vertical bar), which is helpful in generating multisets by recursive functions.

The format of expressions is left undefined, being determined by the underlying language (C++ in the case of LOOPN++).

2.4 Fields

```
field   →  type  ident [ = value ] [ | guard ]     -- data field + value + guard
                 { , ident [ = value ] [ | guard ] } ;
```

A **field** is a class component of some type, optionally with a default initial value, as in:

```
type  ident = value;
```

If the field is local, then every instance of the containing class will associate this default value with the field on initialisation. If the field is exported, then the instantiation of the containing class may specify an alternative initial value for the field, using the notation:

```
type  object = [ident:value, ...];
```

in which case, each instance of the containing class may have a different value for this field.

The type determines the kind of field – a field of (predefined) type `integer` will simply be a constant; a field of (multiset) type `integer*` will be a place holding integer-valued tokens (the container doesn't change but its contents can vary); a field of (user-defined) type *C* could be a subnet instance. Where a field is an instance of a transition class its graphical representation is a rectangle. Where a field is an instance of a state class, its graphical representation is an oval.

Examples of field definitions were shown in figs 2.1, 2.3 and 2.4. Fig 2.4 showed one field *n* which was a constant, and two fields *p* and *q* which were places (since their type was *integer**).

A field specification may also specify a **guard** which is a boolean expression. The interpretation of the guard varies depending on whether the field occurs in a state class or a transition class. For a component of some state class, the guard specifies an **integrity constraint**. If the integrity constraint is ever violated, then the program will abort with an appropriate error message. For a component of a transition class, the guard specifies an **enabling condition**. If the guard is not satisfied, then the transition is not enabled and will not fire. The guard is like a class invariant of Eiffel [35].

2.5 Transitions and actions

```
trans   →  TRANS id [ : [ parent ] {, parent } ]   -- transition definition
             { field }                              -- data fields
             { action }                             -- token + anonymous actions
           END id

action  →  type  ident <- place  [ | guard ]       -- input action + select condition
        →  type  ident -> place  [ | guard ]       -- output action + output value
        →  type  ident -- place  [ | guard ]       -- test action + select condition
        →  procedure-call                          -- interact with environment
```

Traditional Petri Net formalisms include the fundamental concept of a transition. As already noted, OPNs and LOOPN++ build systems from typed components, where each type is predefined or defined by a class. A **transition** is an instance of a transition class. A **transition class** is distinguished from a state class by containing at least one arc (or named action) as an immediate component.

An **action** is the only construct for changing the state of an object. Actions may be input actions, output actions, test actions, or anonymous actions. Apart from anonymous actions (or procedure calls), each action has an associated identifier. A binding of a transition associates a value with each such action identifier. (Note that the grammar allows field declarations in transitions, in which case these fields are also bound to appropriate values in each binding of the enclosing transition.)

The anonymous actions (or procedure calls) of a state class are executed on instantiation of the class. The actions of a transition class are executed each time the transition fires. In order to support formal analysis, anonymous actions must have no effect on the firing of transitions in the net. Graphically, anonymous actions are drawn as annotations of transitions, while named actions are drawn as directed arcs, following the usual Petri net conventions. Textually, each named action may have a guard, while graphically there is only one guard annotating each transition as a whole.

Rather than declaring a class and instantiating it for each transition, syntactic sugar is provided for declaring a transition class with a singleton instance:

```
Trans ident
  type  x <- p | …;                    -- zero or more input actions
  type  y -> q | …;                    -- zero or more output actions
  procedure-calls                      -- zero or more anonymous actions
End ident
```

An **input action** extracts a value from an object and is written in LOOPN++:

```
type  x <- p | guard;
```

where the constraint "`| guard`" is optional. For such an input action to occur, the value of `x` obtained from `p` must be of an appropriate type and satisfy the condition, if any. The value `x` is called a **token** (or tokens), while the object `p` is called an **input place**. The `guard` is a boolean expression which *may* contain terms of the form:

```
x = value
```

which is interpreted as testing that the components of `x` match those specified by the value. For example, the transition of fig 2.4 has an input action which selects an integer token provided its value is less than *n*.

A **test action** examines a value in an object and is written:

```
type  x -- p | guard;
```

where the constraint "`| guard`" is optional. For such a test action to occur, the value of `x` in `p` must be of an appropriate type and satisfy the condition, if any. The `guard` is a boolean expression which *may* contain terms of the form:

```
x = value
```

which is interpreted as testing that the components of x match those specified by the value.

An **output action** deposits a value into an object and is written:

```
type  x -> p | guard;
```

where "| guard" is optional. For such an output action to occur, the value of x must be acceptable to p. Again, the value x is called a **token** (or tokens), while the object p is called an **output place**. The guard is a boolean expression which *will be* of the form:

```
x = value
```

which is interpreted as generating an object with components matching that of the value. For example, the transition of fig 2.4 has an output action which generates a token *y*, a copy of *x*.

The current reference semantics for OPNs dictates that output tokens are always newly-generated objects or newly-generated copies of existing objects. In this way, it is not possible to carry a reference to a remote object (such as a place) around a Petri Net – it is only possible to refer to locally accessible objects. Alternative reference semantics may be possible, but have not yet been investigated.

2.6 Functions

```
func   →  type  ident  ( parms ) [ = fvalue ]    -- function definition
fvalue →  expr                                   -- possible function values
       →  FORALL type ident -- place [ | guard ]; end FORALL
       →  EXISTS type ident -- place [ | guard ]; end EXISTS
       →  COUNT  type ident -- place [ | guard ]; end COUNT
```

A function defines a parameterised expression, which returns a value based on the other features (and hence state) of an object. Functions take parameters and return values of some type (which may be a predefined type, a class type, or a multiset type).

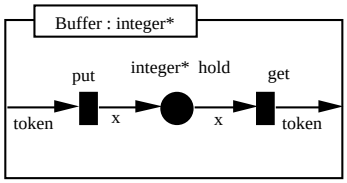
Functions can examine the state of an object, but cannot change that state. Quantification can be used to determine some value based on some or all of the tokens resident in a place. If the result of such functions are to be well-defined, their evaluation cannot be concurrent with transitions which modify the state under inspection. In other words, functions have some aspects of transition semantics and are hence graphically represented by rectangles.

2.7 Places and super places

In OPNs and LOOPN++ a **place**, in the simplest case, is an object of multiset type, declared as in:

```
type*  ident
```

This type implies that a place may have incident arcs, and thus supports input, output and test actions. In general, any class which inherits from a multiset class will support input, output and test actions, and can be instantiated to form a place or, more precisely, a super place. The inherited multiset class determines the type of tokens which can be exchanged with the environment, while the components of the class determine what information is stored and how tokens are exchanged. Graphically, the possible exchange of tokens with the environment is indicated by arcs incident on the class frame. While these arcs may be inscribed by arbitrary multisets, efficient implementation is possible if these arcs exchange one token at a time (here called *token*). Note that this does not mean that a super place may only accept or offer one token at a time, but that each such token offer or acceptance involves one occurrence of an interface transition (here *get* or *put*). Experience seems to indicate that this restriction is acceptable in practice and merely reflects the notion that tokens in a place are independently accessible. Semantically, tokens can be deposited into a super place if the internal activity of the super place accepts exactly those tokens, and conversely for extracting tokens. An example of a super place which will buffer integer tokens is shown in fig 2.5.




```
Class Buffer : integer*
  integer* hold = ...;
  trans put : put
     integer x -> hold | x = token;
  end put
  trans get : get
     integer token <- hold;
  end get
End Buffer
```

Fig 2.5: A super place for an integer buffer

The textual version assumes that the support of a multiset type for input, output and test actions is provided by (pseudo) transitions *get*, *put* and *see*, which serve to define *token* or use it for output. Defining how a super place supports these actions is a matter of inheriting the previous definitions and overriding them with extended definitions.

2.8 Substitution transitions and super transitions

In OPNs, it is possible to emulate the HCPN notion of substitution transitions [17] by defining classes with exported places. When these classes are instantiated, their exported places can be bound to places external to the instance (as in HCPN port assignments).

In this paper, we have followed the HCPN graphical conventions of drawing such substitution transitions as rectangles with incident arcs. However, such conventions are considered to be misleading since they suggest transition semantics, i.e. the synchronisation of the arc actions.

On the other hand, the formal definition of OPNs [27] allows for the notion of super transitions which do have transition semantics. However, the strict synchronisation of arc actions seems to be too restrictive in practice. We are currently investigating a compromise alternative, where the external environment perceives the arc actions to be synchronised, while internally they may be achieved by some sequence of actions [29].

3 Introduction to Z39.50-1992

In this section, we introduce the essential features of the ANSI Z39.50-1992 Standard for Information Retrieval [1], prior to specifying the protocol in the subsequent section. We concentrate on the transition table which formalises the protocol and the identification of the services.

The standard defines the operation of what is called the Z39.50 origin and target, which are those parts of the client and server respectively, which provide the facilities associated with networked information search and retrieval, as shown in fig 3.1.

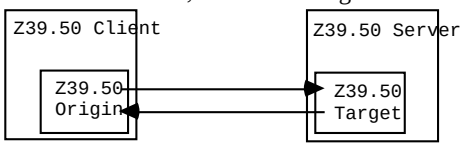


Fig 3.1: Basic structure of the Z39.50 protocol

A Z39.50 session consists of three phases: the establishment of the session, information transfer, and termination. Each phase uses a number of facilities. The establishment phase is handled by the Initialization Facility. The main facilities used during the information transfer phase are the Search Facility (for querying databases at a target), the Retrieve Facility (for retrieving copies of database records) and the Result-set-delete Facility (for deleting result sets known to the target). The termination phase is signalled either by the explicit use of the Termination Facility or the implicit termination by a communication failure or other external event. Other facilities overlay these and handle issues such as access control, accounting and resource control. The role of the origin and target cannot be reversed.

OSI terminology [38] defines protocols as a collection of services and thus the networking standards documents tend to reflect this terminology and this modularisation of protocols. Thus, the Z39.50-1992 Standard speaks of a number of facilities (as already noted), each of which consists of one or more services, as shown in fig 3.2.

Facility	Service
Initialization Facility	Init Service
Search Facility	Search Service
Retrieval Facility	Present Service
Result-set-delete Facility	Delete Service
Access Control Facility	Access-control Service
Accounting/Resource Control Facility	Resource-control Service Trigger-resource-control Service Resource-report Service
Termination Facility	IR-abort Service IR-Release Service

Fig 3.2: The facilities and services of Z39.50-1992

Unfortunately, the modularisation concepts and terminology are not always consistently applied. The Z39.50 standard identifies each service and describes it individually within the standard in unconstrained natural language. It also provides the format of Protocol Data Units (PDUs) in ASN.1 notation. However, the only other description of the protocol is the monolithic state transition table(s) of fig 3.3. It is difficult to separate out the individual services from this state transition table, let alone identifying the normal and abnormal activity of each service. This can only be done by reference to the informal, natural language description of the services. This style of transition table mitigates against modularity.

As well as being deficient with regard to modularisation, the transition table does not even constitute a formal definition. It includes loosely-defined entries such as *stkst* and *popst* for saving and restoring the state. In fact, the standard states: *The IRPM state table does not constitute a formal definition of the IRPM. It is included to provide a more precise specification of the protocol procedures.*

It is clearly desirable for the protocol to be formally specified and for the specification to match the OSI terminology of services. Such modularity will help to achieve an intellectually manageable model or specification. This is done below for the Z39.50-1992 standard (cf. [31, 32]).

Abbreviations					
A	Association control	Iab	IR abort	req	request
Aab	Association abort	ind	indication	resp	response
Acc	Access-control	Init	Initialise	Rsc	Resource-control
Arel	Association release	IR	Information Retrieval	Rsrp	Resource-report
conf	confirmation	Irel	IR release	Srch	Search
Dlte	Delete	Prsnt	Present	Trigrc	Trigger-resource-control

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 + (3)					
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 3.3: Transition table and abbreviations for Z39.50-1992

The transition table specifies the allowable events depending on the current state of the Z39.50 origin. Thus, in the initial or closed state (1), the client may request the establishment of a connection by submitting an *Init req*. In response, the origin sends an *Init PDU* to the target and transfers into state 2. If the target accepts the connection by sending an *Init resp PDU (ACCEPT)*, then the origin notifies the client to this effect with an *Init conf +* and moves into state 3. In this state, the origin is prepared to accept service requests such as *Srch req, Prsnt req*, etc.

4 Basic Services of Z39.50-1992

We now use OPNs and LOOPN++ to model the Z39.50-1992 protocol. We have already noted that the Z39.50-1992 standard formalised the protocol in a monolithic transition table. It has long been recognised [13, 37] that the successful management of complexity requires support for abstraction with clean module boundaries. In this section, we demonstrate how this can be achieved in OPNs by modelling the basic services of Z39.50 (leaving the enhanced services to §5). We consider the passive data components, the generic services, and the composition of those services.

4.1 Passive data – the Z39.50-1992 message formats

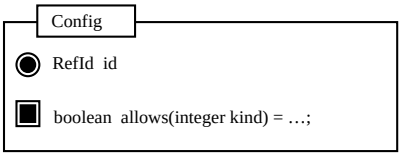
We commence by considering a number of definitions which are commonly required in the protocol description. Firstly, there are a number of constant definitions which distinguish the different kinds of protocol data units (PDUs). These would be defined in a class as in fig 2.1, which could then be inherited by other classes so as to share these common definitions.

Secondly, all PDUs include an optional *Reference-id.* The format of this and its usage are not specified by the standard, except to require that a request PDU including a *Reference-id* must be matched by a response PDU with the same *Reference-id*, and to require that any intermediate Access control or Resource control request must specify the same *Reference-id.* Because of the vagueness of these requirements, our earlier definition of the Z39.50-1992 protocol omitted this optional *Reference-id* [31]. However, its usage is clarified in the Z39.50-1995 standard and is there required for concurrent operations. Accordingly, we define a class *RefId* (as in fig 2.2) with a function to determine if one *RefId* instance matches another. Note that the definition of the function is not supplied, although some simple definition could be given, such as the expression *id=self.* In reality, the function result depends on the way *Reference-id*s are stored, which is not defined by the standard nor in the above class. In a sense, the class serves as an abstract or deferred class, i.e. it will not be instantiated as is. Instead, we will instantiate a subclass which *will* include one or more data fields to store the *Reference-id*. Polymorphism means that these subclass instances can be used anywhere that a superclass instance is specified (see §9.1).

Thirdly, the class for the protocol data units (PDUs) can be given, as in fig 2.3. (The class for messages (MSGs) exchanged with the protocol user will have a similar format.) This class includes a field for the kind of PDU (which will be a value from the class *common* above); it includes the *Reference-id* by instantiating the class *RefId*; and it supplies functions to determine if the PDU is a request, an accepting response or a rejecting response for a certain kind of service with a given *Reference-id*. Note that when the class *PDU* is instantiated, the field *id* can be instantiated to be an instance of some subclass of *RefId*.

Once again, the functions defined for *PDU* are sufficient to determine the behaviour of the protocol entities, and it is a natural object-oriented solution to encapsulate the information in this way. Additional information (specified in the standard for each kind of PDU) will be included in subclasses of *PDU* which can then be used polymorphically in superclass contexts.

Fourthly, before turning to the modelling of active components of Z39.50, namely the protocol services, we note that those services will need to hold the state of the protocol, i.e. the service in progress together with its *Reference-id.* It is also appropriate to retain information about the *allowable* services of the protocol, which are determined by negotiation during initialisation. Accordingly, we define a configuration class to store the *Reference-id* and the set of allowable services as in fig 4.1. Once again it is not necessary to spell out the details of the fields which store the allowable services – it is sufficient to have a function which determines if a service is allowed.

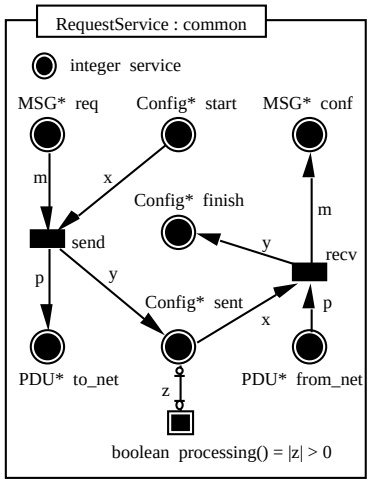



```
Class Config
  Export allows;
  // Configuration data fields
  …
  // Configuration function
  boolean allows(integer kind) = …;
End Config
```

Fig 4.1: Graphical and textual representation of the configuration class

4.2 Modularity – the generic services of Z39.50

We have already noted in the introduction to §4 that the Z39.50-1992 standard identifies the various services of the protocol in the text of the standard, but not in the transition table. 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 4.2). This class can then be instantiated for each of the above services, possibly fusing the initial and final states.



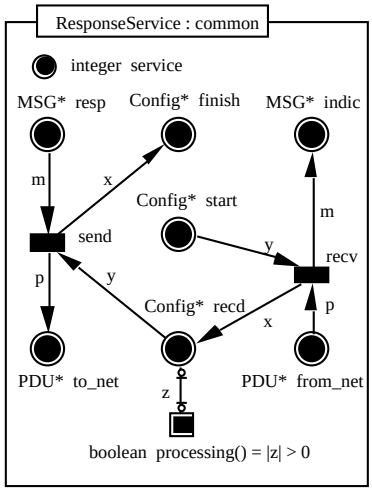

```
Class RequestService : common
  Export service, req, conf, to_net, from_net,
         start, sent, finish, processing;
  integer service;
  Config* start, sent, finish;
 MSG*    req, conf;
  PDU*    to_net, from_net;
  boolean processing() = exists z <- sent;

  trans send
     Config x <- start | x.allows(service);
     MSG    m <- req | m.request(service, x.id);
     Config y -> sent | y = x[id: …];
     PDU    p -> to_net | p = m[…];
  end
  trans recv
     Config x <- sent;
     PDU    p <- from_net | p.accept(service,x.id);
     Config y -> finish | y = x;
     MSG    m -> conf | m = p[…];
  end
End RequestService
```

Fig 4.2: 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 arcs are annotated with token variables. The *send* transition can fire if there is a request message to transmit, and the protocol component is in its initial (*start*) state. The appropriate PDU is sent to the net. The *recv* transition can fire if the appropriate reponse PDU is received from the net and the protocol component is in its intermediate (*sent*) state. The appropriate indication message is sent to the protocol user and the protocol component enters its final (*finish*) state. The function *processing* is defined to return *true* if the place *sent* contains at least one token. Following the conventions of [22], 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 and/or responses, as in fig 4.3. Note that *recv* transition stores a configuration *x* in place *recd* which is the same as *y* but with its *Reference-id* set to the *Reference-id* of the incoming request. This makes it possible to include it with the response.



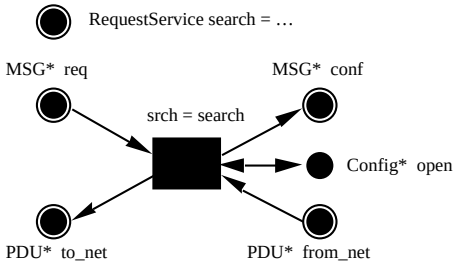

```
Class ResponseService : common
  Export service, resp, indic, to_net, from_net,
         start, recd, finish, processing;
  integer service;
  Config* start, sent, finish;
 MSG*    req, conf;
  PDU*    to_net, from_net;
  boolean processing() = exists z <- recd;

  trans recv
     Config y <- start | y.allows(service);
     PDU    p <- from_net | p.request(service,y.id);
     Config x -> recd | x = y[id:p.id];
     MSG    m -> indic | m = p[…];
  end
  trans send
     Config y <- recd;
     MSG    m <- resp | m.accept(service,y.id);
     Config x -> finish | x = y;
     PDU    p -> to_net | p = m[…];
  end
End ResponseService
```

Fig 4.3: Graphical and textual representation of a response service

4.3 Compositionality – the basic Z39.50 origin entity

The Z39.50 Search facility, like a number of other facilities, can be modelled by instantiating the request service of fig 4.2 as in fig 4.4. Since this is just one component of the Z39.50 origin entity, it has been drawn without a class frame. The textual form (which would appear as an annotation 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 *Config* 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 4.4: Graphical and textual representation of the instantiation of the Search request service

Specifying the data field *search* is sufficient to instantiate the service, complete with the binding of exported components. As observed in §2.8, this is semantically identical to the annotation used in the Design/CPN tool [19]. However, it is traditional for high-level Petri Nets to show the interaction of the service with the places using the graphical transition notation. This notation has been used in this paper, but we consider it somewhat misleading since it does not imply the usual transition semantics (see §2.8).

Similar instantiation of request services for the Z39.50 Present, Delete and Resource-report facilities will cover the entries in part 1 of the state transition table (of fig 3.3) 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, as is done in the combined service of fig 4.5.

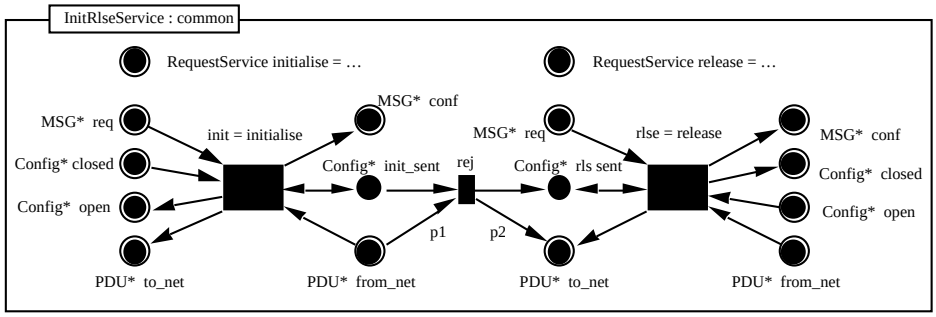


Fig 4.5: Graphical representation of the Initialze and Release services

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, x.id)*.

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 [17], though the ability to specify partially defined components like *RefId* (of fig 2.2) is not. In the following sections, further aspects of the protocol are discussed using OPN facilities which are not readily supported by other Petri Net formalisms.

5 Enhanced Services of Z39.50-1992

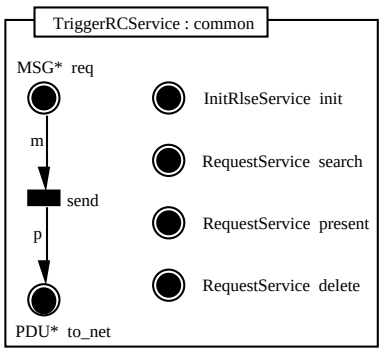
Having considered the basic services of Z39.50-1992, we now consider the enhancements. In the following sections we consider the *Trigger Resource Control* service, *Access Control*, *Termination Control*, and the origin entity including all these enhanced services.

5.1 Weak coupling of subnets – the Trigger Resource Control service

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 [14] advocated place fusion, transition fusion, and invocation transitions. Only the first has been implemented in the widely used package Design/CPN [19] while the second is common in other object-oriented net formalisms [6, 9]. 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 [35] 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 [22] it has been shown that this provision can be formally defined to support the usual step semantics of Petri Nets, and furthermore, nets with such extensions can be transformed into behaviourally equivalent CPNs. Another case study showing the value of this feature can be found in [30].

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 4.2), this has only been exploited in the *Initialize-Release* service (fig 4.5). 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 5.1.




```
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
```

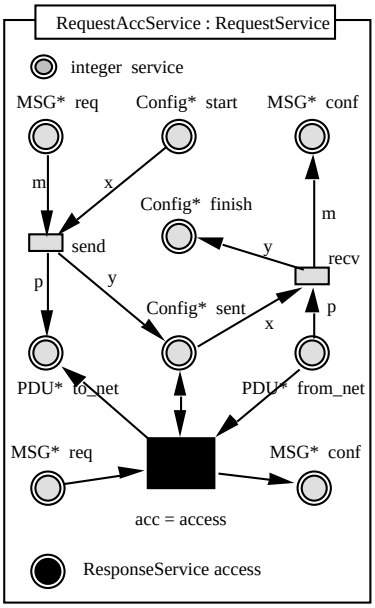
Fig 5.1: Graphical and textual representation of the Trigger-resource-control service

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 simplicity of this interface is possible because of the weak coupling achieved by the use of access functions. What is true of this example is even more imperative for complex systems. As far as possible, the complexity of module interactions should be minimised.

5.2 Incrementality and Access Control

A further demonstration of software reuse enabled by inheritance and polymorphism can be made by considering the provision of access control in the the Z39.50 protocol. This was not examined in §4 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 3.3 that the *Access Control* facility temporarily interrupts a service (with a transfer to state 9).

It is possible to include *Access Control* as net components undifferentiated from the normal operation, but this can all too easily obscure the central function of the service (or facility), as is the case with the monolithic transition table. With OPNs, it is possible (and desirable) to specify these extensions as extensions of the normal service. Thus, the generic *RequestService* of fig 4.2 can be extended to a generic service with access control as shown in fig 5.2. This *RequestAccService* inherits from *RequestService*, with inherited components shown graphically by a grey shading. (The arcs which are inherited without change should also be shaded in grey, but the drawing tool used to prepare this paper did not have that capability.) *RequestService* is augmented with an instance of *ResponseService*, which responds to the Access Control request.




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

Fig 5.2: Graphical and textual version of a request service with access control

Note that we have here displayed the inherited components. This should obviously be a configurable option of a graphical editor, but our experience indicates that where the graphical representation of a Petri Net conveys the pattern of interaction, this information is lost if only the incremental changes are shown (as would be the case in fig 5.2). On the other hand, if the inherited components do not reflect patterns of interaction, then their omission is not a problem (as is the case for *common* in figs 4.2 and 4.3).

An interesting question arises here concerning the *ResponseService* which provides the Access control for each *RequestService*. Should there be one such *ResponseService* for all *RequestService*s or one for each. As shown in fig 5.2, the field *access* is exported. If it is desirable to have one instance for all *RequestService*s, then the environment can bind this field to some global instance. Otherwise, the environment can omit a binding for this field, in which case there will be one instance of *ResponseService* for each *RequestService*. The choice will be constrained by whether the *ResponseService* can uniquely determine the *RequestService* in which it was initiated. That would be possible if *Reference-id*s were always used, but otherwise is problematical.

5.3 Incrementality and Termination Control

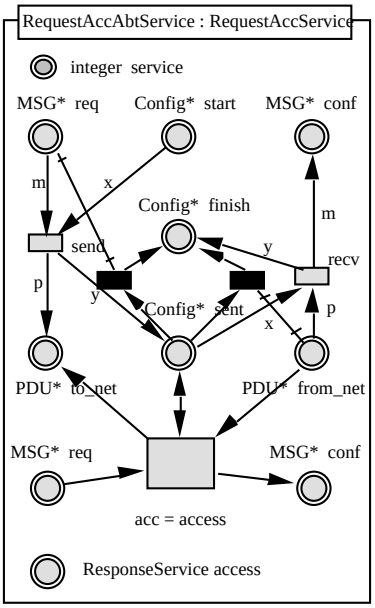
The Z39.50-1992 protocol specifies two services as part of the *Termination Facility* – an abrupt termination or abort service, and a graceful termination or release service. The release service is an origin-initiated, acknowledged service like any other. It has already been considered in fig 4.5. The abort service is different to the others since it is not acknowledged. It can be initiated at any time, and it can be initiated by any party – the origin, the target, or even the underlying Association Control. The abort service must leave the origin entity in the closed state.

The abort service breaks encapsulation – no matter how deeply nested in other services (e.g. within an access control request, within a retrieval request) the origin must transfer into the closed state after having received an abort request. This is apparent in the transition table (of fig 3.3) – the reception of any abort request or indication leads to an immediate transfer into

the closed state (state 1). This is in marked contrast to the *Access control* service which is properly nested, since it causes the current state to be saved on a stack and later restored.

In modelling the abort service as an OPN, it is possible to reflect the breaking of encapsulation by including an additional exported abort state in each of the generic services of §4.2. When the services are instantiated, this abort state could be bound to the closed state. Each service could transfer to this abort state whenever an abort request or indication were received. Thus, no matter how deeply nested within services, every service could immediately return to the closed state in response to an abort request or indication.

Alternatively, the abort service could be modelled in a more encapsulated way. Each service could simply test for an abort request or indication and could then transfer to its final state. Since the service only tests for the abort, it is not consumed. In other words, each level of encapsulation would test the same abort request and transfer to its final state. Eventually, some global transition could consume the abort request or indication and transfer into the closed state. Because we have chosen to emphasise the modularity of our solution, this latter approach is adopted (as in fig 5.3).




```
Class RequestAccAbtService : RequestAccService
  trans abort_request
    Config x <- sent;
    MSG    m -- req | m.request(abort,x.id);
    Config y -> finish | y = x;
  end abort_request
  trans abort_indication
    Config x <- sent;
    MSG    p -- from_net | p.request(abort,x.id);
    Config y -> finish | y = x;
  end abort_indication
End RequestAccAbtService
```

Fig 5.3: Request service with access control and abort

5.4 The enhanced Z39.50-1992 Origin Entity

The Z39.50 origin entity can now be defined as a collection of the above extended services, as shown in fig 5.4. The transitions in the Z39.50 origin, while not annotated, will deal with the abort requests and indications, which are simply examined by the more deeply nested services (see §5.3). In both cases, there is a transfer from the *open* to the *closed* state. In the case of an abort request, the abort is passed to the network. In the case of an abort indication, the indication is passed to the user of the origin entity.

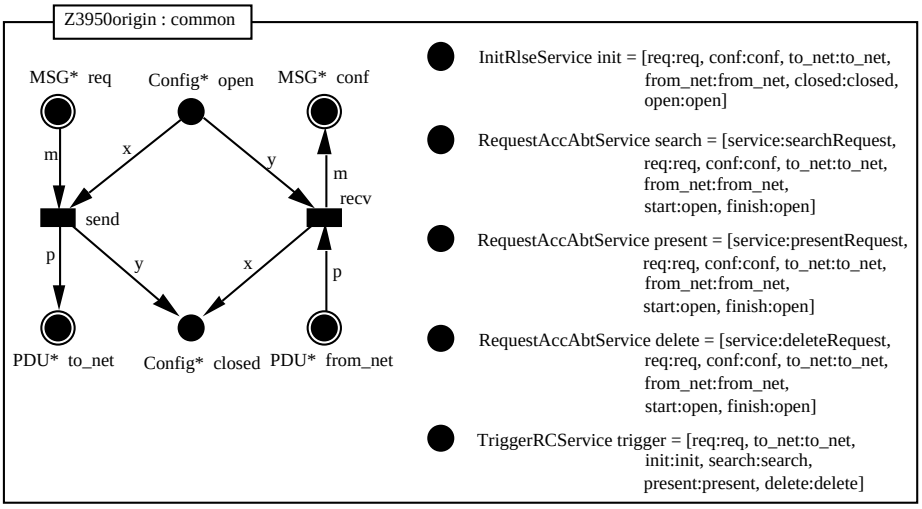


Fig 5.4: The Z39.50-1992 origin entity

Rather than clutter up the diagram of the *Z3950origin*, the super transition abstractions for the services have been omitted, although they could be supplied as in fig 4.4. Note that a number of the services are passed as parameters to the *Trigger-Resource-Control* service, as required (see §5.1).

6 Introduction to Z39.50-1995

Having modelled the Z39.50-1992 protocol, we now turn to the 1995 version to see if it can be modelled as an incremental extension of the 1992 version. In this section we consider the main features of the Z39.50-1995 standard.

The primary aim of the development of Z39.50-1992 was to achieve bit-compatibility with the ISO *Search and Retrieval* standard. Z39.50-1995 was developed to add features to those implemented in Z39.50-1992. Accordingly, Z39.50-1992 replaced and superseded Z39.50-1988, and specified version 2 of the protocol. Z39.50-1995 specifies both version 2 and version 3. In the words of the standard: *Z39.50-1995 is a compatible superset of the 1992 version. An implementor may obtain complete details of version 2 from the Z39.50-1995 document and build an implementation compatible with Z39.50-1992.* This suggests that it should be possible to develop a model of version 3 as an incremental extension of version 2.

The relationship between facilities and services for Z39.50-1995 is given in fig 6.1. The similarities with fig 3.2 reinforces that version 3 operates in much the same way as version 2. The Retrieval Facility has been enhanced to cope with the segmentation of records; the Termination Facility has been simplified; and three additional facilities have been added, Sort, Extended Services and Explain.

The Sort Facility allows for the presentation of records retrieved in a specified order. Scan allows for the interactive browsing of a list or index. Extended Services is used to initiate a specific operation which is executed outside of the Z39.50 session and whose progress may be monitored using Z39.50 services.

Facility	Service
Initialization Facility	Init Service
Search Facility	Search Service
Retrieval Facility	Present Service Segment Service
Result-set-delete Facility	Delete Service
Access Control Facility	Access-control Service
Accounting/Resource Control Facility	Resource-control Service Trigger-resource-control Service Resource-report Service
Sort Facility	Sort Service
Browse Facility	Scan Service
Extended Services Facility	Extended Services Service
Explain Facility	(see below)
Termination Facility	Close Service

Fig 6.1: The facilities and services of Z39.50-1995

The Explain Facility uses the normal Z39.50 facilities to access a special database on the server which then provides details of the server implementation – hours of operation, charges, contact information, databases available, attribute sets or extended services supported. Some of this information is intended for direct display to the user of the client, and others for the internal use of the client.

Significantly for an examination of this protocol, is the introduction of the capacity to handle multiple concurrent operations. There are also other relatively minor, but significant changes, such as support for different character sets, units, and for negotiation on the presence or absence of features.

As with Z39.50-1992, each service is individually described within the standard in unconstrained natural language. However, instead of one monolithic state transition table, there are several (see fig 6.2). These tables are loosely differentiated on function. Table 1 is in three parts, covering the initialisation, processing and termination phases. Table 2 specifies the Present Service (with its possibility of segmented responses), and Table 3 specifies *operations other than Present*. The standard says that these tables describe: *three protocol machines, one for the Z-Association (called the "Z-machine") and two for Z39.50 operations (called "O-machines"). ... There is one instance of the Z-machine (within a given application association) each for the origin and target; there may be multiple concurrent instances of the O-machines.*

Therefore, the modularity is not to enhance understanding, per se, but to show how multiple concurrent operations are achieved within the standard. In order to handle the complexity of concurrent operations, variables are used in the state tables and a syntax for testing these variables has been developed.

Abbreviations							
Acc	Access-control	Init	Initialise	resp	response	Srch	Search
conf	confirmation	Prsnt	Present	Rsc	Resource-control	Trigrc	Trigger-resource-control
Dlte	Delete	req	request	Seg	Segment	Z	Z-association

Table 1, part 1: State Table for Origin Z39.50 Association: Initialization Phase

State / Event	Closed 0	Init sent 1	Acc recvd 2	Rsc recvd 3
Init req	Init PDU; (1)			
Init resp PDU+		Init conf +; setopCnt=0; :[conc] (5) else (4):		
Init resp PDU-		Init conf –; (0)		
Acc PDU		Acc ind; (2)		
Acc resp			Acc resp PDU; (1)	
Rsc PDU		Rsc ind; :[resp] (3) else (1):		
Rsc resp				Rsc resp PDU; (1)

Table 1, part 2: State Table for Origin Z39.50 Association: Processing Phase

State / Event	Serial idle 4	Concurrent idle 5	Serial Active 6	Concurrent active 7	Z-Acc recvd 8	Z-Rsc recvd 9
<op> req	Initiate <op>; (6)	Initiate <op>; (7)		Initiate <op>; (7)	Initiate <op>; set RetSt=7; (8)	Initiate <op>; set RetSt=7; (9)
EndOp ind			(4)	Decr; :[noOps] (5) else (7):	Decr; :[noOps] set RetSt=5:; (8)	Decr; :[noOps] set RetSt=5:; (9)
Z-Acc PDU		Acc ind; set RetSt=5; (8)		Acc ind; set RetSt=7; (8)		
Z-Acc resp					Acc resp PDU; (RetSt)	
Z-Rsc PDU		Rsc ind; set Retst=5; :[resp] (9) else (5):		Rsc ind; set Retst=7; :[resp] (9) else (7):		
Z-Rsc resp						Rsc resp PDU; (RetSt)
Close req	Close PDU; (10)	Close PDU; (10)	Close PDU; KillOps; (10)	Close PDU; KillOps; (10)	Close PDU; KillOps; (10)	Close PDU; KillOps; (10)
Close PDU	Close ind; (11)	Close ind; (11)	Close ind; KillOps; (11)	Close ind; KillOps; (11)	Close ind; KillOps; (11)	Close ind; KillOps; (11)

Table 1, part 3: State Table for Origin Z39.50 Association: Termination Phase

State / Event	Close sent 10	Close Recvd 11
AnyOpPDU	(10)	
Z-Rsc PDU	:[noResp] Rsc ind:; (10)	
Z-Acc PDU	(10)	
Close resp		Close PDU; (0)
Close PDU	Close conf; (0)	

Table 2: State Table for Origin Present Operation

State / Event	Present sent 1	Rsc recvd 2	Acc Recvd 3
Rsc PDU	Rsc ind; :[resp] (2) else (1):		
Rsc resp		Rsc resp PDU; (1)	
Acc PDU	Acc ind; (3)		
Acc resp			Acc resp PDU; (1)
Trigrc req	Trigrc PDU; (1)		
Seg PDU	Seg ind; (1)		
Prsnt resp PDU	Prsnt conf; EndOp ind; exit		

Table 3: State Table for Origin Operation other than Present

State / Event	<op> sent 1	Rsc recvd 2	Acc Recvd 3
Rsc PDU	Rsc ind; :[resp] (2) else (1):		
Rsc resp		Rsc resp PDU; (1)	
Acc PDU	Acc ind; (3)		
Acc resp			Acc resp PDU; (1)
Trigrc req	Trigrc PDU; (1)		
<op> resp PDU	<op> conf; EndOp ind; exit		

Fig 6.2: Transition table and abbreviations for Z39.50-1995

The transition table also uses the following miscellaneous actions:

- Initiate <op>: 1. Initiate an O-machine for operation <op> – table 2 or 3
  2. send <op> PDU
  3. set initial state for operation to 1
  4. if concurrent operations in effect, increment opCnt by 1
- KillOps 1. Immediately terminate all active operations
  2. Any PDUs pertaining to these operations are sent to the Z-machine
- Set <variable> = <x> Set the variable to the desired value
- Decr Decrement the variable opCnt by 1
- Exit Terminate the O-machine
- :[cond] act1 else act2: If *cond* is true then perform action *act1*, otherwise do *act2*
- conc True if concurrent operations in effect, otherwise false
- noOps True if no active operations, otherwise false

It is worth noting that the whole notion of concurrent execution of different tables for the Z-machine and the O-machine is not precisely defined, nor is the communication between them, including the ability of one (the Z-machine) to control another (the O-machine) with operations such as *Initiate <op>*, *EndOp ind*, and *KillOps.*

7 New services of Z39.50-1995

Like many protocols, Z39.50 has evolved over time with the addition of further functionality. In this section we consider that additional functionality and how it can be modelled as extensions of the 1992 protocol.

7.1 Additional services and facilities of Z39.50-1995

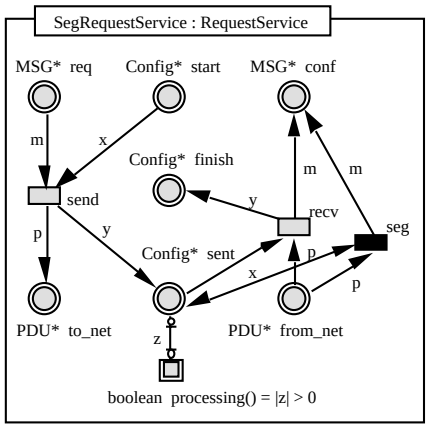
Z39.50-1995 introduces two new services – *Scan* and *Sort*. The former is used to scan terms in a list or index. It is currently the only service in the Z39.50 Browse facility. The latter is used to sort a result set. Both of these services follow the paradigm of the *Search* service discussed in §4.3. They therefore require no further attention beyond the observation that the functions to test for an appropriate response will need to be redefined.

Z39.50-1995 also introduces two new facilities – *Explain* and *Extended Services*. The former allows a client to retrieve details of the server implementation. The latter is used to initiate a specific extended service task, which is executed outside of the Z39.50 session and whose progress may be monitored using Z39.50 services. The beauty of polymorphism is that these new facilities can be encompassed in the existing model without further change.

7.2 Extended service of Z39.50-1995

A significant modification of Z39.50-1995 over the 1992 version is the support for segmentation of retrieval responses, a modification of the *Present* service. Once a search has taken place, the result set is established. The actual records can then be retrieved. Given the variability of the length of the records, it is possible to return multiple records in each response (if the records are relatively short), or only part of a record (if the records are relatively long). Provision for the first is referred to as level 1 segmentation, while additional provision for the second is referred to as level 2 segmentation. Both are accomplished by the target sending a number of *Segment* requests followed by the *Present* response. (These are referred to as *Segment* requests rather than responses in order to fit the request–response paradigm.)

The modelling of this extended service is a simple matter of defining a refined request service as shown in fig 7.3. Note that the refinement simply adds a new transition. The enabling of this transition depends on whether segmentation is supported, an issue which is resolved by negotiation.




```
Class SegRequestService : RequestService
  trans seg
     Config x <- sent | x.allows(Segment);
     PDU    p <- from_net | p.request(Segment,x.id);
     Config y -> sent | y = x;
     MSG    m -> conf | m = p[…];
  end
End SegRequestService
```

Fig 7.3: Enhanced request service for segmented Present

Note that this enhancement has been shown on top of the *RequestService* (of fig 4.2) so as not to clutter the diagram. It could equally have been shown on top of the *RequestAccService* (of fig 5.2), or preferably the *RequestAccAbtService* (of fig 5.3), since it is an enhancement over what has been done before. This approach is quite adequate for our purposes. Another possibility would be to model each enhancement as a separate class and then form the enhanced services by multiple inheritance from the original service and the enhancements.

8 Other extensions to Z39.50-1995

Z39.50-1995 also introduced what it refers to as *Miscellaneous enhancements*. We consider two of these here, both of which do not so much add new functionality but modify the existing protocol scheme. Both of them can be seen as primarily impinging on the state of the connection.

8.1 Negotiated services

Z39.50-1995 modifies the functionality of the 1992 version by extending the negotiation of supported services. Depending on this negotiation, some services will never be used as part of a session. This is simply an extension of the negotiation included in the 1992 version, and the *Config* class (of fig 4.1) will need to be suitably extended.

8.2 Concurrent operations

Perhaps the most significant change introduced by Z39.50-1995 is the support for concurrent operations. This adds significant complexity to the transition table (as already noted in §6).

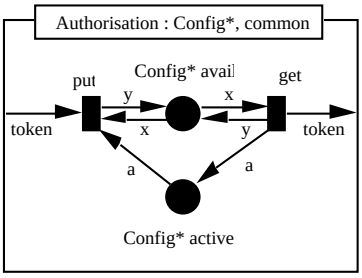
In considering how this extension can be addressed, we observe that in the first instance, the existing services are not modified. A *Search* service still has the same sequence of operations. It needs to have access to the the token in the *start* place (fig 4.2) which is bound to the *open* place (of fig 4.4). The difference is that the access to this token should not prohibit other services from similarly accessing this token. Clearly what has changed is the logic associated with accessing a token from this place.

The logic necessary for concurrent operations can be achieved by replacing the *open* place with a super place which has its own internal logic to control the access to tokens. For serial operations, only one token will be offered. For concurrent operations, multiple tokens will be available. This flexibility would not be available under an alternative definition of super places, which is currently under investigation [29]. Here, a super place has an associated abstract marking. Such a definition is more restrictive but does promise the possibility of more efficient analysis. (We return to this in §10.)

A further implication of the concurrent operations is that the *Reference-id*s specified in the 1992 standard now play a much more significant role. However, the support built into the earlier model is sufficient for our requirements here. In other words, the service definitions of §4 and §5 can be used here without change for concurrent operations. This is quite remarkable considering the magnitude of the changes required in the transition table to support concurrent operations. We believe that this supports the claim that the object-oriented structuring mechanisms supplied by OPNs and LOOPN++ are both powerful and flexible.

As a result of these arguments, the *open* place will be a super place which will offer and accept tokens of type *Config* (which indicate the supported services). Initially, the token deposited in the *open* place by the *Init* service needs to indicate the services which have been negotiated. Subsequently, it can be handled as shown in fig 8.1.

The intention is that place *avail* holds a single *Config* token which determines (see §4.1) the allowable services which have been negotiated. (The initial value will indicate that no services are allowed.) The place *active* will hold one token for each activated service. (The initial value will indicate that the only active service is initialisation.) The tokens in place *active* are copies of the tokens which have been offered to the environment. They are matched when the tokens are returned by the environment. They are therefore largely redundant, but in some situations, it will be helpful to have them collected together in this net. For example, the reception of an Access control request should be targetted at the active operation specified in the *Reference-id* in the PDU. If there is no such operation, the Access control is taken to relate to the Z-association. It will be necessary to be able to test this condition by comparison with the *Reference-id*s of all active operations.




```
Class Authorisation : Config*, common
  Config* avail = ...;
  Config* active = ...;
  trans put : put
     Config a <- active | a = token;
     Config x <- avail | a.matches(x);
     Config y -> avail | y = x[...];
  end put
  trans get : get
     Config x <- avail;
     Config y -> avail | y = x[...];
     Config a -> active | a = x[...];
     Config token = a;
  end get
End Authorisation
```

Fig 8.1: Authorisation super place

Each time a *Config* token is removed from this super place, the token in place *avail* is modified to reflect the currently allowable services. Similarly, when such a *Config* token is deposited in this super place, the token in place *avail* is again suitably modified. A complex function can be used to compute the modified setting. For example, if only serial operations are allowed, then any removal of a *Config* token will result in the token in place *avail* indicating no allowable services. On the other hand, if concurrent operations are allowed, the removal of a *Config* token may result in no change to the token in place *avail*. A more subtle example is that the reception of an Access control PDU for the Z-association will prohibit the reception of further such Access control PDUs, until the current one has been dealt with.

8.3 Termination facility

A significant change in Z39.50-1995 concerns the *Termination facility*, so much so that it is difficult to know how the 1995 version can be described as a compatible superset of the 1992 version. The 1992 version supported both abrupt termination (the abort service) and graceful termination (the release service), while the 1995 version only supports a form of termination with both graceful and abrupt features. It is graceful because it is acknowledged. It is abrupt because all currently active operations are abandoned.

There are also more subtle differences. For example, if an initialisation request is refused in Z39.50-1992, a release request is sent to the association and eventually confirmed. In Z39.50-1995, no such release request is sent. Again, in Z39.50-1992 abort indications can be received both from the information retrieval service and from the association, while in Z39.50-1995, only a close PDU can be received.

This means that some functionality of the 1992 version needs to be removed. OPNs and LOOPN++ do not support the removal of functionality in a subclass. This is a significant problem and demonstrates that not all evolution of protocols fits appropriate patterns. To some extent, this problem can be hidden by extending the negotiation of services to ensure that the 1992 termination facility is never enabled.

Once the above problem has been identified and resolved, the handling of termination in Z39.50-1995 can be addressed, largely because of the modular approach adopted in §5.3. Firstly, the *InitRlseService* of fig 4.5 needs to be modified so that the rejection of the *Initialise* request results in a transfer to the closed state, and not the transmission of a *Release* request, in line with the transition table of fig 6.2. Secondly, the new transitions of fig 5.3 should be responses to *Close* rather than *Abort* requests and indications. Finally, the origin entity of §5.4 needs to be modified to ignore (possibly delayed) PDUs, other than the final *Close* request or indication (in line with the transition table of fig 6.2 – Table 1 part 3).

9 Other Object-Oriented Features

This section highlights some aspects of OPNs which are considered important for modelling network protocols, either in the example above of Z39.50, or in possible extensions of the example.

9.1 Support for inheritance and polymorphism

Languages which support the definition and instantiation of modules are classified as *object-based* [45]. Such languages encourage a certain amount of software reuse. Thus, in §4 and §5 the Z39.50 protocol was built up as a number of services, each of which instantiated one of two classes. Such multiple instantiation comes under the title of object-based and is commonly found in Petri Net formalisms such as Hierarchical Coloured Petri Nets (HCPNs) [17].

One of the primary motivations for the development of object-oriented technology was the quest for more effective mechanisms for software reuse [35]. This has primarily been achieved through inheritance and polymorphism, which qualify a language (or formalism) to be described as *object-oriented* [45]. Inheritance allows a class to derive its features from another (its parent class), and then to augment 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 [3, 44, 46].

OPNs and LOOPN++ support inheritance and polymorphism, which can be used to structure the models and to benefit from this style of software reuse. This has been extensively used in the modelling of the Z39.50 protocol above. It made possible the minimal definition of the classes *RefId*, *PDU* and *Config* in §4.1. These were defined with sufficient information to cater for the required operations, but with the understanding that an actual implementation would attach additional information. This is particularly apparent for the class *PDU.* For the purposes of the protocol (figs 4.2 and 4.3), it is sufficient for this class to have functions which determine whether an instance is a request, an acceptance, or a rejection of a particular kind of service. In reality, a *PDU* will carry extensive information pertinent to the service request or response, but this is not relevant to the operation of the protocol.

9.2 Support for genericity

One of the primary motivations for the development of the OSI Reference Model [15] 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 should perform the same 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 (generic) 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 §9.1. 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. Another example would be the *RequestService* of fig 4.2 which receives tokens of type *MSG* from the higher layer and submits tokens of type *PDU* to the lower layer. So far, we have assumed that these classes are the same, but it would be more common to find that the class *PDU* augments the class *MSG* with some control information.

The requirements in such situations can be highlighted by examining typical transitions which would add and delete such header information for a data link layer, as in fig 9.1.

```
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 9.1: 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)* <: *class(y)* means that the declared class of *x* is a subclass of that of *y*
(c) given *class(x)* <: *class(y)*, we write *class(x) = class(y) + C*, where *C* is the class containing those components which augment *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* <: *class(x)*
(f) *class(y') = class(y) + C* <: *class(y)*

(where the operator + has higher priority than = and <:) 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. Thus, 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++ [35, 40] 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 [36], which allows *any* component class to be consistently renamed while still retaining type safety. An alternative approach, which is worth investigating, is that adopted by the BETA programming language, where the concept of virtual function components is extended to include the notion of virtual class components [34].

9.3 Support for mobile objects

Recent years have seen an increasing interest in distributed object systems and the use of mobile objects. The unified class hierarchy of OPNs readily supports the modelling of such systems – tokens may instantiate arbitrary classes, and class components may also instantiate

arbitrary classes. Thus, in §4.1 the Z39.50 protocol data units were defined by classes encapsulating both data fields and functions. Elsewhere [27], we have used OPNs to model the documents of a hypothetical Electronic Data Interchange system [20], where the documents encapsulated transitions, in addition to fields and functions.

10 Analysis of OPNs

As noted in the introduction, the amenability of Petri Nets to automated analysis has been a key aspect which has facilitated their beneficial application to the development of reliable network protocols. The introduction into OPNs of powerful object-oriented structuring primitives has not yet been matched by appropriate analysis techniques. In fact, this is considered to be a significant area of research in which only the preliminary steps have been taken. For example, the design of the OPN formalism has been constrained so as to be able to transform OPNs into behaviourally equivalent CPNs [23]. It is anticipated that this will provide the foundation for adapting existing CPN analysis techniques for OPNs.

There are two forms of analysis commonly applied to high-level Petri Nets. Reachability analysis generates the graph of all possible reachable states and the transitions between those states. This form of analysis is normally plagued by the problem of state explosion where the number of possible states is exponential in the size of the net model. Techniques have been developed to reduce the size of the reachability graph without losing the essential properties of the system. Invariant analysis, as its name suggests, determines some property which is invariant over the execution of the net. A place invariant specifies a weighted marking which does not change over the firing of any (enabled) transition in the net, while a transition invariant specifies a weighted step which leaves the marking unchanged.

In this section, we highlight some of the issues which will need to be addressed in providing appropriate analysis techniques for OPNs. We consider the general issue of modular analysis and observe how this has lagged behind the development of modular specification techniques. We then consider the specific issues pertinent to reachability analysis and invariant analysis. Finally, we consider the ideal goal of developing incremental analysis techniques.

10.1 Modular analysis

Hierarchy constructs were first proposed for CPNs in 1990 [14]. Of these, only substitution transitions and place fusion were implemented in HCPNs [17, 19]. The behaviour of these nets was then defined in terms of the expanded CPN (after the substitution transitions were appropriately expanded and the relevant places fused). In other words, the modular definition of HCPNs was not matched by the notion of modular behaviour or modular analysis. It is our contention that the effective analysis of complex systems modelled by Petri Nets will increasingly demand the development of modular analysis techniques.

As far as CPNs are concerned, we note that symmetry analysis [18] can be most effective in reducing the size of the reachability graph by collapsing symmetrical states onto the one state. This technique seems most appropriate to CPNs where colours have been used to fold similar subnets onto the one subnet. Similarly, stubborn set analysis [41] has been used to reduce the number of possible interleavings of independent concurrent processes which are covered by the reachability graph. These two techniques are orthogonal and can be extremely effective in reducing the size of the reachability graph, but neither of them is particularly relevant to modular analysis.

The development of modular analysis techniques has lagged behind the development of hierarchy constructs for CPNs. However, proposals have been made recently for the modular analysis of Modular Coloured Petri Nets (MCPNs) [10, 11]. For place invariants [10], the weighted marking is specified on a module-by-module basis. Provided the weight functions are consistent (i.e. they agree on fused places), and provided that they constitute a place flow for each module, then the local weight functions determine a global weight function which

constitutes a global place flow for the whole net. For reachability analysis [11], it turns out that modular analysis is much simpler where transition fusion is used as opposed to place fusion. Here, the computation of the state space (or reachability graph) for the individual modules is interleaved with the computation of a synchronisation graph, which captures the interaction points of the various modules, i.e. the firing of fused transitions.

While these modular analysis techniques are important contributions, we believe that their results will be limited because of the overly general form of modular nets that they address. A net is formed from a number of modules which are combined using the arbitrary mechanisms of place fusion and transition fusion. The modules are not constrained to have any particular properties, nor are the results constrained to place-bounded or transition-bounded modules. We believe that better results will be obtained by restricting attention to super places and super transitions, i.e. modules which have place or transition properties. Some preliminary steps in this direction have already been made [29].

A different approach to modular analysis has been developed using process algebra techniques [42]. Here, process algebra reduction techniques are applied to place-bounded subnets so that the same externally visible behaviour is obtained from simpler subnets. Clearly, this approach has great relevance to complex systems modelled by modular Petri Nets.

10.2 Reachability analysis

As already noted, reachability analysis involves the generation of every reachable state and every possible transition between these states, and consequently is severely affected by having the number of states being exponential in the size of the model. One simple implication of this, which is not unique to OPNs, is that predefined types need to be restricted to finite subranges. Thus, while *integer* type fields have been used in the specification of the Z39.50 protocol (e.g. for the PDU *kind* in fig 2.3), only a small number of values is actually used, and this should be made explicit.

OPNs introduce another potential problem area with the use of object identifiers to identify or address the various objects or class instances. The formal definition of OPNs specifies one set of object identifiers per class [27], and these sets are potentially infinite, because the number of instances of any class is potentially infinite. In order to make reachability analysis computable or tractable for OPNs, the sets of object identifiers will need to be finite (as in [5]). It will also be desirable (if not necessary) to include some form of symmetry analysis [18], so that states differing only with respect to their labelling by object identifiers are treated as identical.

10.3 Invariant analysis

As already noted, invariant analysis identifies some property which is invariant over the firing of the net. Modular invariant analysis [10] can be extended to OPNs [25], but the weight functions cannot be statically determined for each class instance, since the number of instances varies over the life of the net. Instead, the weight function for a class instance is formed from the composition of a weight function for the class and a weight function determined by the context of the module instance (relative to the root class).

Simpler invariant results can be obtained if classes are constrained to define super places and super transitions, i.e. subnets with place and transition properties [29]. This follows since those place and transition properties are captured by weight functions and local invariant properties. For example, the abstract marking of a super place is defined by a weighted marking of the associated subnet. This should be invariant over the internal activity of the subnet. Consequently, a place invariant over the abstract net (where super places are treated as places) can be extended to a place invariant over the expanded net.

10.4 Incremental analysis

Just as it is important to develop modular analysis techniques for modular nets, so the ultimate goal in analysing OPNs is to develop incremental analysis techniques to match the incremental modelling capabilities. The ideal is that when one net component is replaced by a more refined component, then the analysis results can be suitably modified rather than recomputed from scratch.

There are many issues to be addressed here, not the least of which is the specification of what behaviour is preserved from a parent class in the subclass. Some proposals have required a form of bisimilarity, while other have required a state preorder [5]. Our experience with some practical case studies indicates that both of these are too restrictive [26].

In some cases, we anticipate that incremental analysis will be relatively straightforward. For example, the additional support for *Segment* requests (in fig 7.3) and even the addition of *Access Control* (in fig 5.2) do not detract from the previous behaviour but only extend it – the added components only respond to different kinds of PDU. The original state sequences are retained, though possibly augmented (as in the state preorder property of [5]).

In other cases, we anticipate that incremental analysis will not be possible. For example, the modified support for termination control (as discussed in §8.3) is so different to the earlier version that it is difficult to see it as an incremental change.

Finally, there are some cases where it is problematic whether incremental analysis will be possible. For example, the addition of an abort capability (in fig 5.3) means that the original sequences of states are still possible, but parts may now be bypassed.

11 Conclusions

This paper has introduced Object Petri Nets, both in their graphical form and in their textual form (in the language LOOPN++). There is an economy of notions in OPNs, particularly with respect to the unified class hierarchy. Thus, classes encompass simple predefined types, user-defined classes without actions, and user-defined classes with actions. Consequently, 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. The unified class hierarchy therefore caters directly for the arbitrary nesting of objects, which sets OPNs and LOOPN++ apart from other object-oriented net formalisms [4, 6, 9, 39, 43].

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. As is standard in object-oriented languages, subclass instances can be used polymorphically in superclass contexts. This makes it possible to introduce minimal definitions into classes (as in §4.1) knowing that more complex definitions can be substituted later without affecting the logic of the net. In other words, inheritance can be used to structure the model of the protocol, and also to capture the different configurations and the evolution of the protocol.

OPNs support both super places and super transitions. A super place can accept or offer tokens to its environment under the control of its internal logic. Since the token type of the interface is determined by the multiset class that it inherits, polymorphism means that such super places can be substituted for simple places in a net. Similarly, a super transition has transition characteristics, but the extent to which this is the case varies between different Petri Net formalisms and is the subject of further investigation (see §2.8).

In summary, OPNs incorporate a uniform and flexible type system which provides good support for modularity, inheritance, polymorphism, and mobile objects. This paper has

demonstrated how these attributes can be of significant benefit in the incremental modelling of network protocols. The possibility of incremental modelling is seen as one of the chief benefits of OPNs. It allows the modeller to capture the basic services of a protocol and clearly identify the enhancements. It allows the identification of different protocol configurations, and it allows the evolution of protocols to be captured in the model.

Perhaps the most dramatic demonstration of this was the way that the support for concurrent operations, introduced in Z39.50-1995, could be captured with minimal change to the previously developed net. The only thing required was to override a simple place with a super place, the super place encapsulating the logic of when multiple concurrent operations could be allowed.

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. As noted in §10, this is considered to be a first step in adapting CPN analysis techniques for use with OPNs. We also noted that the development of better modular analysis techniques will also be of great importance to OPNs. Clearly, the ideal will be to develop incremental analysis techniques to match the incremental development possibilities of OPNs. A related issue is the kind of behaviour that should be preserved from a parent class in a subclass – it is unclear whether existing proposals are adequate for practical applications.

Currently, a preliminary version of LOOPN++ has been implemented [28]. 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, we have 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 [30]. Another variant of this compiler produces Java code, which could therefore be used in the development of web applications [33].

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] ANSI *Information Retrieval (Z39.50): Application Service Definition and Protocol Specification* ANSI/NISO Z39.50-1995 (version 3), American National Standards Institute (1995).

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

[4] 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).

[5] E. Battiston, A. Chizzoni, and F. de Cindio *Inheritance and Concurrency in CLOWN* Proceedings of Workshop on Object-Oriented Programming and Models of Concurrency, Torino, Italy (1995).

[6] 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).

[7] 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).

[8] 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).

[9] 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).

[10] S. Christensen and L. Petrucci *Towards a Modular Analysis of Coloured Petri Nets* Application and Theory of Petri Nets, K. Jensen (ed.), Lecture Notes in Computer Science 616, pp 113-133, Springer-Verlag (1992).

[11] S. Christensen and L. Petrucci *Modular State Space Analysis of Coloured Petri Nets* Application and Theory of Petri Nets, G.D. Michelis and M. Diaz (eds.), Lecture Notes in Computer Science 935, pp 201-217, Springer-Verlag (1995).
[12] 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).
[13] 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).
[14] 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).
[15] ISO *Information Processing Systems – Open Systems Interconnection: Basic Reference Model* International Organisation for Standardization and International Electrotechnical Committee (1984).
[16] 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).
[17] 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).
[18] K. Jensen *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use – Volume 2: Analysis Methods* EATCS Monographs on Theoretical Computer Science, Springer-Verlag (1994).
[19] K. Jensen, S. Christensen, P. Huber, and M. Holla *Design/CPN™: A Reference Manual* MetaSoftware Corporation (1992).
[20] P. Kimberley *Electronic Data Interchange* McGraw-Hill (1991).
[21] 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).
[22] 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).
[23] C.A. Lakos *Object Petri Nets – Definition and Relationship to Coloured Nets* Technical Report TR94-3, Computer Science Department, University of Tasmania (1994).
[24] C.A. Lakos *From Coloured Petri Nets to Object Petri Nets* Proceedings of 16th International Conference on the Application and Theory of Petri Nets, Lecture Notes in Computer Science 935, pp 278-297, Torino, Italy, Springer-Verlag (1995).
[25] C.A. Lakos *The Object Orientation of Object Petri Nets* Proceedings of Workshop on Object Oriented Programming and Models of Concurrency, Torino, Italy (1995).
[26] C.A. Lakos *Pragmatic Inheritance Issues for Object Petri Nets* Proceedings of TOOLS Pacific 1995, pp 309-321, Melbourne, Australia, Prentice-Hall (1995).
[27] C.A. Lakos *The Consistent Use of Names and Polymorphism in the Definition of Object Petri Nets* Proceedings of 17th International Conference on the Application and Theory of Petri Nets, Lecture Notes in Computer Science 1091, pp 380-399, Osaka, Japan, Springer-Verlag (1996).
[28] C.A. Lakos *The LOOPN++ User Manual* Technical Report R96-1, Department of Computer Science, University of Tasmania (1996).
[29] C.A. Lakos *On the Abstraction of Coloured Petri Nets* Proceedings of 18th International Conference on the Application and Theory of Petri Nets, Lecture Notes in Computer Science 1248, Springer-Verlag (1997).
[30] 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).
[31] C.A. Lakos, J.W. Lamp, C.D. Keen, and B.W. Marriott *Modelling Network Protocols with Object Petri Nets* Proceedings of Workshop on Petri Nets Applied to Protocols, pp 31-42, Torino, Italy (1995).
[32] J.W. Lamp *Encoding the ANSI Z39.50 Search and Retrieval Protocol using LOOPN* Honours Thesis, Department of Computer Science, University of Tasmania (1994).
[33] G.A. Lewis *Producing Network Applications Using Object-Oriented Petri Nets* Honours Thesis, Department of Computer Science, University of Tasmania (1996).
[34] O.L. Madsen and B. Møller-Pedersen *Virtual Classes: A Powerful Mechanims in Object-Oriented Programming* Proceedings of OOPSLA 89, SIGPLAN Notices 24, pp 397-406, New Orleans, Louisiana, ACM (1989).
[35] B. Meyer *Object-Oriented Software Construction* Prentice Hall (1988).
[36] J. Palsberg and M.I. Schwartzbach *Object-Oriented Type Systems* Wiley Professional Computing, Wiley (1994).
[37] D.L. Parnas *On the Criteria to be Used in Decomposing Systems into Modules* CACM, **15**, 12, pp 1053-1058 (1972).
[38] M.T. Rose *The Open Book: A Practical Perspective on OSI* Prentice-Hall (1990).
[39] 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).

[40] B. Stroustrup *The C++ Programming Language (Second Edition)* Addison-Wesley (1991).

[41] A. Valmari *Stubborn Sets for Coloured Petri Nets* Proceedings of 12th International Conference on the Application and Theory of Petri Nets, Aarhus (1991).

[42] A. Valmari *Compositional analysis with place-bordered subnets* Proceedings of 15th International Conference on the Application and Theory of Petri Nets, Lecture Notes in Computer Science 815, pp 531-547, Zaragoza (1994).

[43] 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).

[44] J.M. Vlissides *Generalized Graphical Object Editing* Technical Report CSL-TR-90-427, Stanford University (1990).

[45] P. Wegner *Dimensions of Object-Based Language Design* Proceedings of OOPSLA 87, pp 168-182, Orlando, Florida, ACM (1987).

[46] A. Weinand, E. Gamma, and R. Marty *ET++ – An Object-Oriented Application Framework in C++* Proceedings of OOPSLA 88 Conference, ACM (1988).