Analysis and Design for Concurrent Object Systems

Derek Coleman, David Skov
Intelligent Networked Computing Laboratory
HP Laboratories Bristol
HPL-93-18
February, 1993

object-oriented analysis and design, concurrent objects, concurrency, industrial applications

This paper describes OO/CAD, a systematic method for developing concurrent object-oriented industrial applications. These applications often adopt an explicit concurrency model with two abstraction levels, objects and threads. Existing object-oriented methods have an inappropriate model of concurrency. OO/CAD method produces a concurrency architecture which takes run time platforms into account. Output from the method is compatible with other existing object-oriented methods which then can be used for further design of each sequential system component.

Internal Accession Date Only

© Copyright Hewlett-Packard Company 1993

1 Introduction

This paper describes a method for developing object-oriented applications that involve concurrency. The method, called OO/CAD (*Object-Oriented Concurrency Analysis and Design*), is useful for systems which need to be structured as a network of sequential processes. This includes many types of applications, including real time software and firmware for controlling instruments and computer peripherals.

The paper starts by reviewing the requirements for OO/CAD. The next section describes the concurrency model that is assumed for the method. Later sections present an overview of the method and show its application to a non-trivial example. The paper concludes with a discussion.

2 The Needs of Industrial Software Developers

The increasing use of software in instruments means that concurrency is becoming a common place feature of many industrial applications. Furthermore, the need to continually improve software productivity and quality is encouraging the uptake of object technology. As a result, it is becoming more important to be able to develop concurrent object-oriented software in a systematic fashion.

At the start of our work, a number of these projects were surveyed from different Hewlett-Packard divisions, [12]. They were asked about the nature of the concurrent applications that they developed. All the applications exhibited a similar limited form of concurrency characterised by:

- processes are sequential C++, or C, programs.
- concurrency is provided by the operating system.
- a relatively small number of processes, usually less than 30.
- very limited use of dynamic process creation.

The projects also emphasised the lack of guidance from existing object-oriented methods on how to obtain a concurrent structure for applications. The following requirements were identified for an object-oriented method for these kinds of applications. These were that the method should be:

- object-oriented in approach, and use familiar object-oriented notions wherever possible.
- complementary to the use of existing object-oriented methods for developing the processes.
- of help finding concurrency, and provide heuristics for improving efficiency and checks for safety etc

- manageable, with a defined process and deliverables that can be checked by inspections etc.

The next section introduces the concurrency model that is used in OO/CAD.

3 Concurrency model

Concurrency models for object-oriented systems can be partitioned into *explicit concurrency models* and *implicit concurrency models* [14].

The *implicit concurrency model* integrates parallelism into the object structure. Processes, i.e control activities are encapsulated within objects. The concurrency commitment for a system is decided by defining activity behavior for objects.

The *explicit concurrency model* describes concurrency external to the object structure. The parallelism sits on top of the object structure rather than being integrated into it. Explicit mechanisms like locks, monitors and semaphores are used to ensure object integrity. The explicit model requires two abstraction levels, objects and threads, and makes it the responsibility of the analyst to separately specify the parallel activity.

Most concurrent object-oriented languages support the implicit model. A few languages support the explicit model by having a set of predefined threads or root object types for initiating parallel activity. Systems in which the concurrency is provided by the operating system, rather than the programming language, conform to the explicit model. The systems surveyed in the previous section all use explicit concurrency. The processes can be considered as objects as they encapsulate their data which can only be accessed via the interface. Therefore, for the rest of the paper we use the term *Concurrent Object* to denote a system component which has a thread in the overall parallel system behaviour. Since the concurrency is provided by the operating system there are no classes and no inheritance. OO/CAD uses an explicit concurrency model without inheritance and classes for concurrent objects.

The method develops a system architecture in terms of a set of synchronous communicating concurrent objects, each representing a single thread of control. Communication is in terms of *events*. Events are atomic units of mutually synchronised communication that may carry data from the sender to the receiver. They can be used to model a number of different styles of concurrent object communication such as one to one and multiple receives and sends. Multiple participation in the same event gives concurrent object synchronisation. Asynchronous communication can be modelled by the use of a concurrent buffer object.

4 Overview of the OO/CAD method

The OO/CAD method covers the analysis and design of the concurrent aspects of object-oriented systems. The starting point is a natural language requirements document. The finishing point is a concurrent architecture that takes the run time platform into account.

The method employs three means for finding concurrency:

- *Temporally independent input events.* If there is no ordering on the occurrence of input events then they can be processed by different concurrent objects.
- *Object responsibility.* If an object has to carry out a task that can be performed asynchronously then a new concurrent object may be introduced to take on the responsibility.
- *Shared objects* can be encapsulated in concurrent objects.

Figure 1 shows how the techniques are used in OO/CAD. The nodes correspond to steps for producing and refining the concurrency architecture and object model. As the figure suggests, there is a preferred ordering on the step, but they can be applied iteratively. The

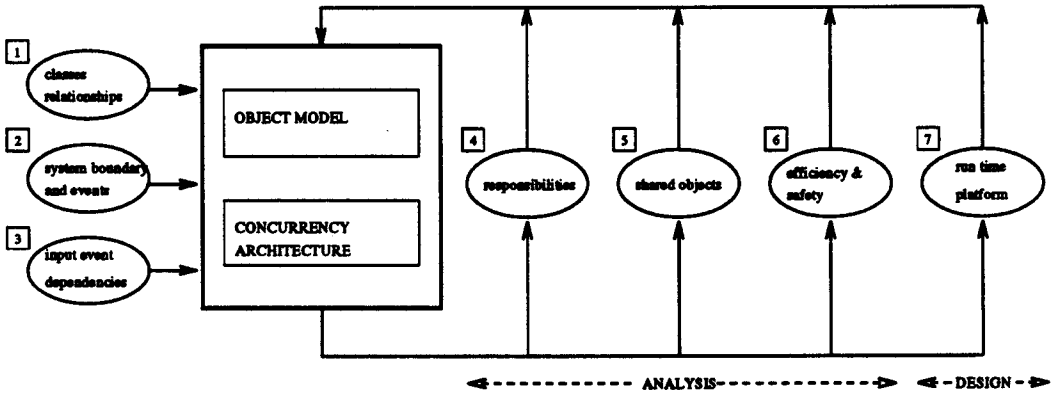

Figure 1: The OO/CAD method

figure also shows the principal deliverables from OO/CAD:

Concurrency Architecture which structures the system as a set of communicating concurrent objects. Each concurrent object is defined by its

- *interface*, i.e. the events it can receive and send
- *life cycle*, i.e. the allowable sequences of events in which the object can participate
- *functionality*, i.e. the responsibilities or tasks that it must perform.

Object Model which shows the static structure of each concurrent object as a set of classes and their relationships.

The first three steps of the method build the first cut at the models; the remaining steps refine them. Step one to six constitutes the analysis phase; step seven is the design phase. We now briefly explain each of the steps in turn.

Step 1: Classes and Relationships This step produces the initial version of the object model, which describes the static structure of the concurrent objects. It provides a vocabulary for expressing the analysis and design.

Step 2: System Boundary and Events The first step in determining the concurrency partition is to find the events that cross the boundary between the system and its environment. An *event-list* is produced for all the events that the system sends and receives.

Step 3: Input Event Dependencies The temporal dependencies between events can be used to determine the first cut at the concurrent architecture of the system. Events which are temporally independent can be handled by different concurrent objects.

Step 4: Responsibilities of Concurrent Objects This step takes system functionality into account in order to discover further concurrency. The CRC notion of responsibility, [1], is used to define the essential tasks that a system has to perform.

Step 5: Shared Objects The responsibilities are investigated to find if there are objects that need to be accessed by more than one concurrent object. These *shared objects* are potentially a source of concurrency.

Step 6: Review of Efficiency and Safety This step completes the analysis phase by applying heuristics to improve efficiency and checks for safety, fairness and liveness properties. It produces a *logical* concurrency architecture that ignores implementation issues, *cf* essential models in structured methods for real time systems [13].

Step 7: Run Time Platform This is the design step that produces a *real* concurrency architecture by mapping the logical concurrency architecture onto the target run time environment.

The next section introduces a problem which is used to explain the method in more detail.

5 Problem Statement

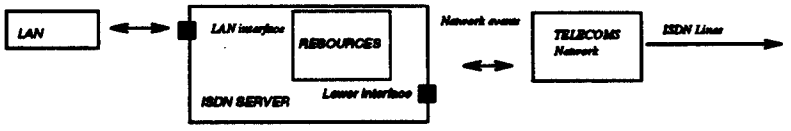

Figure 2: The ISDN Server

The task is to design a simplified Integrated Services Digital Network (ISDN) server. The server is responsible for transmitting packets of data between a Local Area Network (LAN) and a telecoms network; see figure 2. The server receives and sends a number of different kinds of packets and events, as it is shown in figure 3. Packets received from the telecoms

Direction	Identifier	Source/Sink	Description
In-event	LAN-packet	LAN interface	Packet from the LAN
In-event	connect-confirm	Lower interface	Telecom confirmation of connection established
In-event	connect-indication	Lower interface	Telecom request for connection
In-event	disconnect-indicatior	Lower interface	Telecom initiated clearing of connection
In-event	data-indication	Lower interface	Telecom data inwards
out-event	Out-packet	LAN interface	Data packet to be sent to LAN
out-event	connect-request	Lower interface	Server request to establish a connection
out-event	connect-response	Lower interface	Server response that connection is established
out-event	data-request	Lower interface	Telecom data outwards

Figure 3: Packets and events sent and received by the ISDN server

network are sent straight out to the LAN. Packets coming from the LAN carry data and a destination address.

In order to transmit packets there must be a connection. Each connection requires the use of some server resources, *ie* circuits and protocols. An *outbound* connection allows packets to be sent out to the telecom network. It is established by the server requesting the telecom network to connect to a destination address and requires access to configuration data internal to the server. An *inbound* connection permits packets to be received from the telecom network. It is initiated by a request from the the telecom network;

the server is not informed of the address of the source of the packets.

For outbound connections the sequence of events at the Lower interface is: *connect-request* followed by *connect-confirm.* For inbound connections the sequence is *connect-indication* followed by *connect-response*, The server clears the connection when it receives a *disconnect-indication* event.

Outbound connection policy Packets received from the LAN interface are queued separately for each destination address. For each destination the server knows whether there is a connection established or not. If a connection is established, packets are transmitted from the queue. If a packet is received for a destination for which there is no connection, then the server makes the connection as soon as the resources are available. The following rules apply:

1. The server attempts to establish a connection to the destination address.
2. Outbound packets for a destination are queued whilst waiting for confirmation of a connection to that destination.
3. When the connection confirmation arrives, packets can be transmitted and received on that connection.

Inbound connection policy: The server operates only one incoming connection at a time. Incoming data-indication packets are passed to the LAN interface. On receiving an incoming *disconnect-indication* the corresponding destination is disconnected and the associated resources are freed.

6 Method description and Problem solution

Each step in the method is explained and applied to the ISDN server.

6.1 Step 1: Classes and Relationships

The first step produces the initial version of the object model, which is an entity-relationship model for the system and its environment. The relationships include specialisation/generalisation (*is-a*) and aggregation (*part-of*). The object model is a static description of the problem domain concepts and provides a vocabulary for the rest of the analysis and design. In this step, no attention is paid to how the system is partitioned into concurrent objects. Heuristics for constructing an object model can be gleaned from other methods, e.g. Fusion or OMT.

ISDN server: In an object model, classes are represented by rounded rectangles and relationships by diamonds and arcs. The *is-a* relationship is shown as a triangle. Figure 4 models *inbound* and *outbound* connections as specialisations of the *connection* class. A *LAN packet* is an aggregation class containing *data* and a *destination address*. The relationship *sends data via* models the queues that are associated with each destination address.

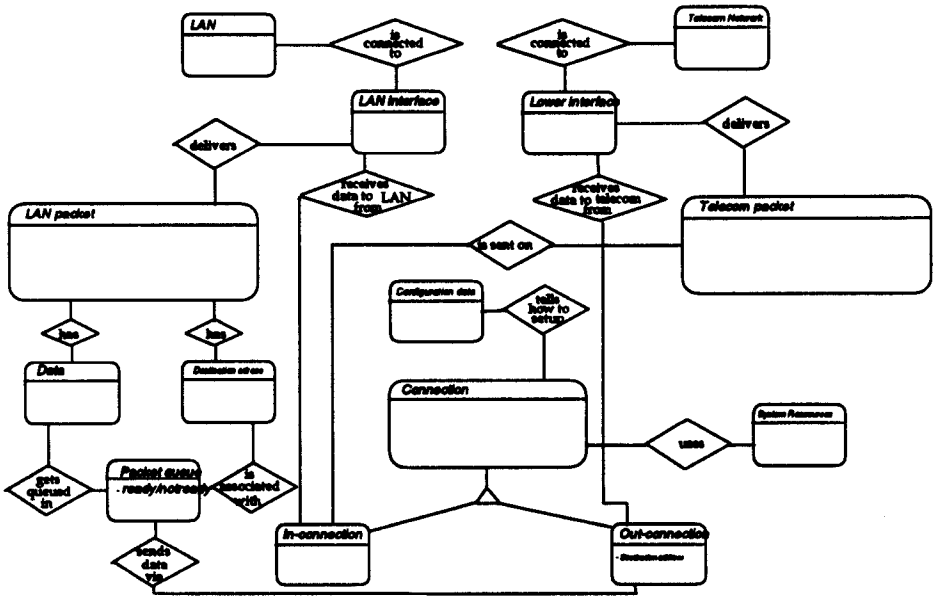

Figure 4: Object Model for the ISDN Server (Step 1)

6.2 Step 2: System Boundary and Events

The step finds the events that crosses the boundary between system and environment. An *event-list* is produced for all the events that the system sends and receives.

A useful technique for discovering the events is to consider different scenarios in which the system can be used. A scenario is a time line diagram, that shows the temporal order of events that result from typical uses of the system. This technique is sometimes called "use-case" analysis and is the basis of the Objectory method [9].

ISDN server: The scenario in figure 5 shows how the server is used to transmit packets to

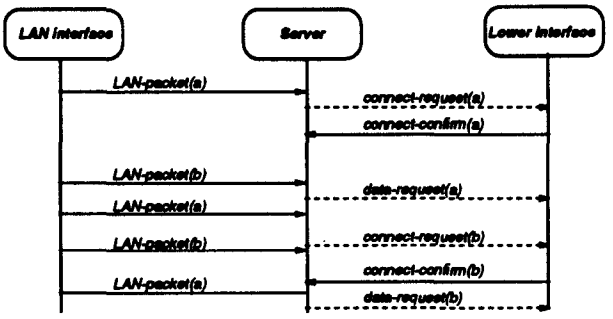

Figure 5: Scenario for transmitting packets to destination addresses a and b (step 2)

two different destination addresses *a*, and *b*. It asynchronously sets up connections towards the destination address while receiving *LAN packets* and transmitting telecom packets (*data-request*) on established connections. The full list of events is constructed by considering other scenarios for different uses of the server. For example, another usage would be transmitting packets from the telecom network. The event list for the problem is contained in the problem statement, figure 3.

6.3 Step 3: Input Event Dependencies

The step produces the first cut at the concurrent architecture of the system. Events which are temporally independent can be handled by different concurrent objects. The temporal ordering of events is determined by building event *life cycles*. Each life cycle is a regular expression which shows the sequencing and choices between events. Life cycles are built by considering the dependencies between in-coming events. Each life cycle is allocated to a separate concurrent object [8].

ISDN server: Two concurrent objects are identified, namely `P1` and `P2`, figure 6. The in coming events related to each of the two concurrent objects are temporally dependent, i.e they happen in the order described in the life cycles below:

```
P1 = LAN-packet_i; connect-confirm_i
P2 = connect-indication_i;data-indication_i;disconnect-indication_i
```

Input events are annotated with *i*, output events with *o*. The initial life cycle for `P1` states: A LAN packet must arrive before a confirmation about a connection setup is received.

The initial life cycle for P2 states: A connection must be established before data can be transmitted into the server from the telecom side. The connection can be disconnected afterwards. For the rest of the example we only consider properties of P1, i.e the connections and data coming from the telecom network are not analysed further.

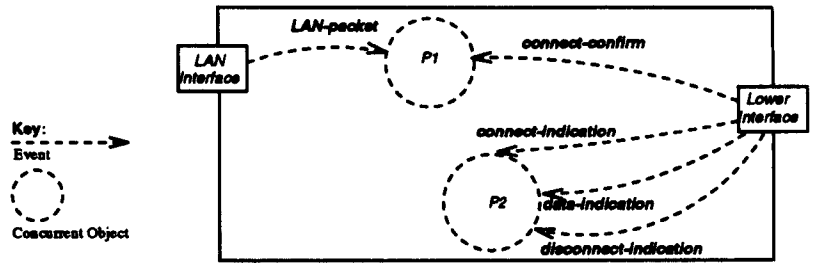

Figure 6: Initial Concurrent Objects with events (step 3)

6.4 Step 4: Responsibilities of Concurrent Objects

In deciding concurrency is it necessary to take system functionality into account. At this early stage in the development it is inappropriate to invest in detailed specifications of functionality. Instead, the CRC notion of responsibility, [1], is used. A *responsibility* is an informal statement of some essential task that the system has to perform. Responsibilities identify problems that have to be solved.

6.4.1 Allocate responsibilities to objects

Each responsibility has to be coupled, *i.e.* be undertaken, by a concurrent object. The coupling can either be *tight* or *loose*. A *tightly* coupled responsibility has to be completed before the concurrent object can process any more incoming events. A responsibility that is *loosely* coupled to a concurrent object may be performed simultaneously with the concurrent object, although it may have to synchronise with some later event.

The CRC game for finding and allocating responsibilities to classes can be adapted for the purpose of finding responsibilities and coupling them to concurrent objects. Each responsibility is documented by describing what the task is, what events coming into the system causes the responsibility, what events may be sent out of the system or to other concurrent objects and whether the coupling is tight or loose

6.4.2 Refine Responsibilities

Responsibilities are a further source of concurrency. In the logical concurrency architecture all responsibilities have to be tightly coupled to some concurrent object. A loosely coupled responsibility must be refined into constructions of other concurrent objects, tight responsibilities and possible asynchronous communication links. Tightly coupled responsibility can also be refined by the introduction of new concurrent objects.

The refinements introduce new concurrent objects and events. The life cycle associated with each existing concurrent object must be updated to show output events. Life Cycles have to be established for each concurrent object, as was done in the previous step. Generally there are many possibilities for refinement. These include decomposing a complex responsibility into simpler responsibilities and allocating them to new concurrent objects. The decisions taken in this step depend very much on the system requirements and the developers experience of developing concurrent systems.

ISDN server: Figure 8 shows the allocation of responsibility to P1. Figure 7 shows the responsibility descriptions.

Responsibility	Description	Causes	Sends	Coupling
Queue Packet	Queues all LAN packets. Packets are queued uniquely for each destination address. A new queue can be created	*LAN packet*	—	Tightly coupled to P1
Check connection	Checks a requst list for whether a connection request has been sent for the actual destination address	*LAN packet*	—	Tightly coupled to P1
Setup Out connection	Allocates resources and sets up connections for transmission of packets to the telecom.	*LAN packet*	*connect-request* to the Lower interface	Loosely coupled to P1. The system must be able to receive LAN packets without delays from connection setup
Send Packets from Queue	Sends data from queues whose destination address are connected. The data is sent to the Lower interface	*LAN packet* *connect-confirm*	*data-request* to the Lower interface	Loosely coupled to P1. The system must be able to send and receive packets at the same time

Figure 7: Responsibility descriptions (step 4)

The loose responsibilities are refined according to the following ideas: (See figure 8).

The *Setup Out connection* responsibility is loosely coupled to P1. We choose to make a new concurrent object P1-2 which gets responsibility for setting up connections towards the telecom network. Every time P1-2 must setup a connection it gets a *create* event from P1-1. A buffer is introduced for the *create* events. The buffer gives an asynchronous timing between P1-1 and P1-2.

The *Send Packets from Queue* responsibility is loosely coupled to P1. We choose to make a new concurrent object P1-3 which gets responsibility for sending packets to the telecom network. Every time a new connection is ready for transmission P1-3 is told by a *connection – ready* event from P1-2. The timing of P1-2 and P1-3 is decoupled by a buffer. P1-3 also gets the responsibility for marking the packet queues when their connections are ready.

New life cycles for P1-2 and P1-3 are created. The life cycle for P1 is updated to a life cycle for P1-1. Events out of the system are mixed into the different life cycles:

```
P1_1  =  (LAN-packet_i;[create_o])*
P1_2  =  (create_i;connect-request_o;connect-confirm_i;connection-ready_o)*
P1_3  =  (connection-ready_i | data-request_o*)*
```

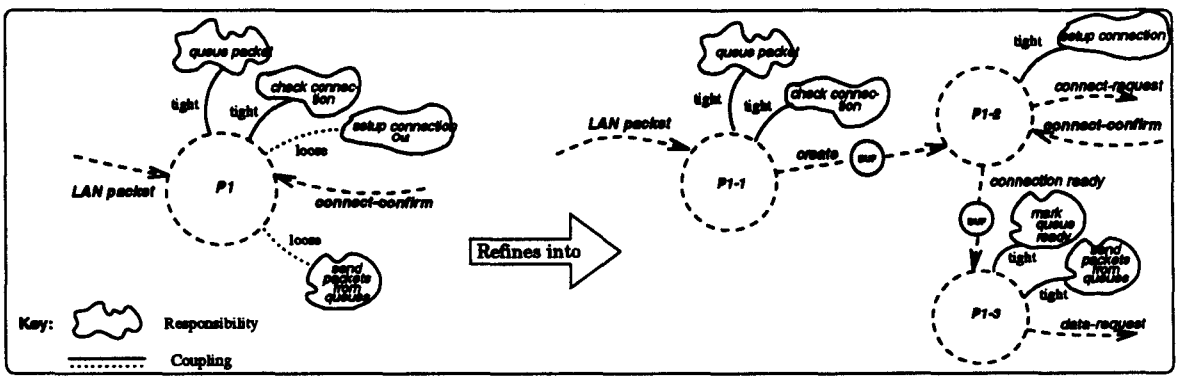

Figure 8: Refinement of responsibilities for initial concurrent objects (step 4)

The life cycle for `P1-1` denotes: After arrival of a *LAN packet* the system will possibly send a *create* event. This pattern is repeated (*). The life cycle for `P1-3` allows for later design to decide how many $data - request_o$ to send out before accepting a *connection-ready* event. The life cycles describes the temporal order of events coming in and out of the system.

The new concurrent objects and the refinement of the responsibilities gives changes in the description of the responsibilities. The changes are described in figure 9.

Responsibility	Description	Cause	Sends	Coupling
Setup Out connection	Allocates resources and sets up connections for transmission of packets to the telecom.	*create*	*connection-ready* to P1-3	Tightly coupled to P1-2 . This implies that P1-2 might have to wait for freeing of resources
Mark queue ready	Takes care of marking that a new queue is ready to be transmitted from	*connection-ready*	—	Tightly coupled to P1-3
Send Packets from Queue	Sends data from queues which are marked ready for transmission	*connection-ready*	*data-request* to the Lower interface	Tightly coupled to P1-3

Figure 9: Updated responsibilities (step 4)

6.5 Step 5: Shared Objects

In this step, the responsibilities and object model are investigated to find if there are objects that need to be accessed by more than one concurrent object. These *shared objects* are potentially a source of concurrency.

6.5.1 Identifying Shared Objects

An object accessed by a responsibility must be an instance of some class in the object model. If this is not the case, then the object model is deficient and must be updated. Each responsibility is considered in turn and the objects that it needs to read and to update are identified. An object is shared if it is accessed by responsibilities coupled to different concurrent objects. The shared objects are added to the concurrency model so as to show read and update access by responsibilities.

6.5.2 Making Shared Objects Concurrent

Access to a shared object is controlled by making it a concurrent object. The protocols for accessing each concurrent shared object need to be designed. For example, consider a protocol for read access to a shared object. One protocol would be to require the reader to send a request data event to the shared object, and then have the shared object send an event carrying the data back. This protocol allows the reader to initiate the transmission of data. A different protocol would give the shared object the initiative by have it send a data event to the reader whenever it was ready. Clearly the two protocols give rise to different behavioural characteristics.

The synchronisation of events must also be defined. For example, read access of shared objects must not happen simultaneously with an update access. The granularity of data associated with each event is a further issue. A balance has to be struck between the computation required to generate or process a packet and the number and frequency of packets to be transmitted. If the components of a shared aggregate object are accessed by different objects, then it may be possible to make each component into a concurrent object.

These issues cannot be properly decided in a local context. They all have to be considered for their impact on system performance. Buhr, [3], has some useful heuristics for this aspect of concurrency design.

Once the decisions have been made, the life cycles of the concurrent shared objects must be defined and the life cycles of existing concurrent must be updated.

ISDN server: Each responsibility is considered in terms of *Reads* and *Changes*, figure 10.

Responsibility	Reads	Changes
Queue packet	—	packet queue's new packet queue's
Check connection	Request list	—
Setup Out connection	system resources	system resources connection's Configuration data
Mark queue ready		packet queue's
Send packets from queue	packet queue's	packet queue's

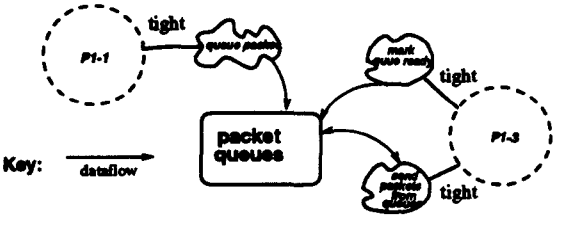

Figure 10: Shared Objects (step 5)

By referring to the Object Model do we identify *packet queues* as shared, see figure 10. The Object Model must be updated with a *Request List* read by the *Check connection* responsibility. The *packet queues* requires **new**, **enqueue**, **dequeue** and **mark ready** as abstract operations. They must be mutually exclusive.

The concurrent object which must encapsulate *packet queues* needs event interface and a life cycle which enforces mutual exclusion of the operations. The event interface decides

whether *packet queues* shall be passive or active in the sense of sending data on requests or by initiating it itself. We propose that *packet queues* decide when to send data. This requires a small change in the event interface and life cycle for P1-3. Below is shown event interfaces and life cycles for all concurrent objects in the system. The system is depicted in figure 11

```
P1_1  =  (LAN-packet_i;[new_o|enqueue_o];[create_o])*
P1_2  =  (create_i;connect-request_o;connect-confirm_i;connection-ready_o)*
P1_3  =  ((connection-ready_i;mark_o) | (dequeue_i;data-request_o)*)*
packet queues = (new_i|((enqueue_i|mark_i);dequeue_o*))*
```

The life cycle for **packet queues** leaves it open how many **dequeue** events should follow (**enqueue|mark**). This will be decided when the *packet queues* is designed.

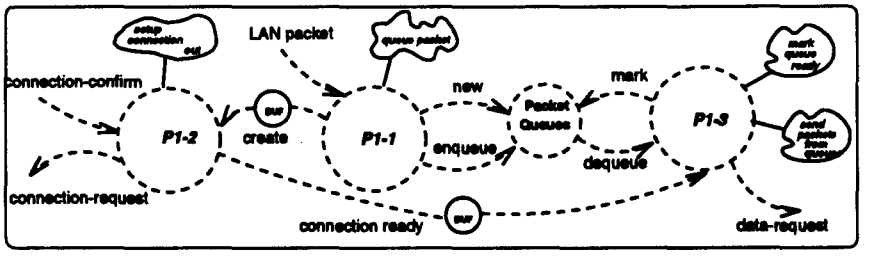

Figure 11: Concurrency Architecture with shared concurrent objects (step 5)

6.6 Step 6: Efficiency and Safety

This step completes the logical concurrency architecture. It is produced by reviewing the logical concurrency architecture that has been produced so far. Heuristics and checks for the following issues need to be applied:

- *Efficiency improvement.* For example, consider merging concurrent objects and/or adjusting the granularity of data passed by events. Consider real time properties by working out end to end responses on events coming into the system. The ADARTS method contains many useful guidelines for this stage, [7].
- *Safety, fairness and liveness properties.* The system should be checked for deadlocks etc. See [4] for guidelines and tests to avoid these problems

The *logical concurrency architecture* now consists of concurrent objects, life cycles, event interfaces and *tightly* coupled responsibilities. The parts of the object model relevant to each concurrent object can be found by examing the responsibilities of each concurrent object.

ISDN server: Figure 11 shows that concurrent object P1-3 *marks queues ready* and receives data via the **dequeue** event. It is more efficient to collapse *packet queues* and P1-3 into

one concurrent object. The activity related to **mark** and **dequeue** events becomes internal for the collapsed concurrent object. This simplification is depicted in figure 12. Buffers are introduced between P1-1 and the collapsed concurrent object for decoupling of timing. The life cycle for the collapsed concurrent object is:

```
P1_3 + pack  =  (connection-ready_i|new_i|enqueue_i|data-request_o*)*
```

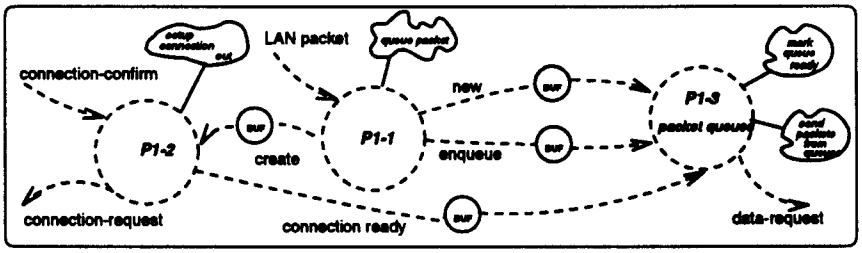

Figure 12: Final logical Concurrency Architecture (step 6)

Figure 12 shows the reviewed final structure of the system. This model is also basis for considering communication overhead on communication links. Safety properties of the system can be derived from analysis of the model. This properties are not further analysed in the example.

6.7 Step 7: Runtime Platform

This step maps the Concurrency Architecture onto a specific run time platform.

Concurrent objects map into the tasks or processes provided by the runtime platform. The responsibilities, object model and life cycle of each concurrent object are the basis for further design. They are a good staring point for any of the object-oriented methods.

Event communication Synchronous event communication has to be implemented by the communication primitives of the target platform. These primitives can be exchanges, pipes, message queues, events etc. The asynchronous buffering has also to be implemented using features of the platform.

Shared Concurrent Objects Although they can be mapped into tasks or processes they can also be mapped to passive implementation platform features, such as monitors, for managing shared memory.

Time critical concerns Time critical concurrent objects in the *logical concurrency architecture* are considered. The facilities offered by the operating system for manipulating the scheduling and priorities of its concurrency units should be used to meet the needs.

The above lists some of the problems in considering a specific runtime platform. The main concern is to ensure that the way of using the operating system primitives does not violate the logical properties of the system.

ISDN server: We do not consider a specific platform in detail for the ISDN server. However, it is a straight forward task to map the Concurrency Architecture into a real time operating system, e.g *pSOS*.

7 Internal Design of Concurrent Objects

It is a requirement that OO/CAD be compatible with object-oriented methods for designing sequential systems. On conclusion of the analysis and design phases of OO/CAD each concurrent object in the concurrency architecture has a description comprising a *event interface*, *life cycle*, *responsibilities* and part of the *Object Model*. Figure 13 shows the deliverables for concurrent object `P1-1`. The deliverables from OO/CAD are expressed in terms which

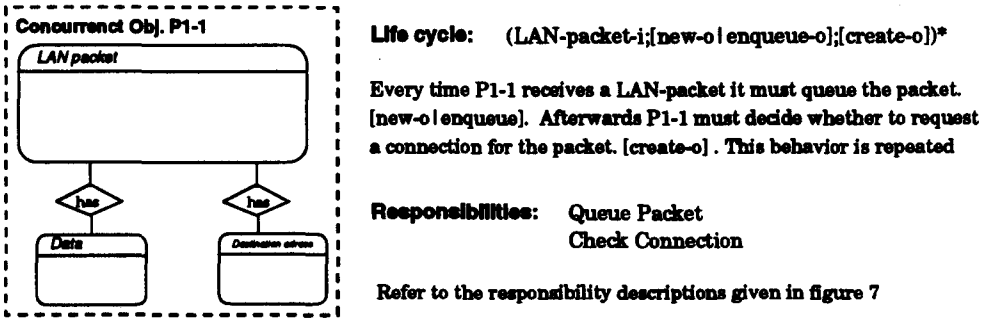

Figure 13: Deliverables for P1-1

are similar to many methods. For example, Coad-Yourdon [5], Fusion [6], OMT [10] and Shlaer-Mellor [11] all use entity-relationship based object models. Likewise the responsibilities and events help define the behaviour to be analysed and designed. Consequently it is straightforward to apply an object-oriented method to the task of designing the concurrent objects.

8 Discussion

The OO/CAD method was developed, because other object-oriented methods, such as Booch [2], OMT and Shlaer-Mellor, do not work well for explicit concurrent systems. These methods adopt the implicit model of concurrency and integrate concurrency into the object structure. The problem, is that there is no straightforward way of mapping complex implicit concurrent structures on to the simple concurrency provided by run-time platforms. Consequently the concurrent aspects of the design have to be done in an *ad-hoc* fashion, which makes them difficult and error-prone.

The OO/CAD method should be judged against the requirements, presented in section 2. To summarise, these were that the method should be object-oriented in approach, assist in finding concurrency, be complementary to the use of object-oriented methods for developing the processes, and be manageable, with a defined process and deliverables. We believe that OO/CAD satisfies all of these. OO/CAD is based on object-oriented concepts, *e.g.* entity-relationship modelling and responsibilities. The method embodies three powerful techniques of finding concurrency. However, more work needs to be done in providing heuristics for ensuring efficiency. Use of OO/CAD does not prejudice the choice of method for designing the sequential object-structures. Object models, responsibilities and event interfaces can be used as the starting point for almost any object-oriented method. The systematic nature of the OO/CAD process, and the well-definedness of its deliverables, means that it is supportive of good project management techniques. One weakness, however, is that OO/CAD does not provide much help with checking whether a concurrent implementation satisfies its requirements document. This a hard research problem.

Currently OO/CAD should be regarded as a "lab prototype". It has been used in experiments, but has yet to be tested by live projects. Controlled technology transfer of OO/CAD is the next step. In Hewlett Packard, we expect to see it used as an supplementary technique for projects using the Fusion method.

Finally it is worth noting that OO/CAD has been developed by focusing on the needs of a particular group of software engineers. The great benefit of this approach is that "customers needs" provide the methodologist with strong criteria for making technical decisions. This is very useful when dealing with highly complex and varied domains such as concurrent systems. We believe that producing software methodologies targeted for particular needs is a useful alternative to the all purpose generalised methodology. In the future, we can imagine methods being produced for applications developers working on particular platforms or in specialised application domains. The future direction of our work will be to develop specialised methods for specific needs, *e.g.* TMN in telecommunications software, distributed object platforms etc.

9 Acknowledgements

The authors wish to thank their colleagues, Chris Dollin and Paul Jeremaes for their contributions to this work. Thanks are also due to Fiona Hayes of HP Germany, Mike Ives and Tony Dicolen of HP USA, and Koichi Nakagawa of YHP Japan for their help in explaining what the problems are.

10 References

[1] K. Beck and W. Cunningham. A laboratory for teaching object-oriented thinking. In *ACM OOPSLA '89 Conference Proceedings*, 1989.

[2] G. Booch. *Object-Oriented Design with Applications.* Benjamin Cummings, Redwood City, CA (USA), 1991.

[3] R.J.A. Buhr. *Practical Visual Techniques in System Design: with Applications to Ada.* Prentice-Hall International, Englewood Cliffs, NJ (USA), 1991.

[4] K.M Chandy and J. Misra. *Parallel Program Design.* Addison Wesley, 1988.

[5] P. Coad and E. Yourdon. *Object-Oriented Analysis - Second Edition.* Yourdon Press, Englewood Cliffs, NJ (USA), 1991.

[6] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Fusion: An object-oriented analysis and design method.* Prentice-Hall International, Englewood Cliffs, NJ (USA), 1993.

[7] H. Gomaa. Structuring Criteria for Real Time System Design. In *Proceedings of the 10th International Conference on Software Engineering, 1988*, pages 418–427, 1988.

[8] B.Graf H. Rischel and A.P. Ravn. *Konstruktion af Formaalsbundne systemer.* Teknisk Forlag, 1987.

[9] I. Jacobson. *Object-Oriented Software Engineering.* Addison Wesley, Reading, MA (USA), 1992.

[10] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall International, Englewood Cliffs, NJ (USA), 1991.

[11] S. Shlaer and S.J. Mellor. *Object Lifecycles: Modeling the World in States.* Yourdon Press, 1992.

[12] D. Skov. Analysis and design techniques for concurrency. Technical Report HPL-93-17, Hewlett-Packard Laboratories, Bristol, Filton Road, Stoke Gifford, Bristol (UK), February 1993.

[13] P.T Ward and S.J. Mellor. *Structured Development for Real Time Systems.* Yourdon Press, 1985.

[14] B.B Wyatt, K. Kavi, and S. Hufnagel. Parallelism in object-oriented languages:a survey. *IEEE Software*, pages 56–65, November 1992.