Chapter 9

Issues and Tools for Protocol Specification

A system is said to be distributed when it includes several geographically distinct components cooperating in order to achieve a common distributed task.

The development of informatic networks and telematic services, as well as the access to public data transmission networks raised the question of building distributed applications. However it should be understood that distributed processing is not only the result of the combination of local data-processing and transmission facilities.

- With the introduction of dedicated processors (eg. I/O processors...), then of multiple-processor systems and finally of local area networks, distributed processing has been introduced inside data-processing systems themselves.
- Simultaneously distributed processing was appearing within intimity of the telecommunication networks themselves, first with packet and circuit networks, second in Common Signalling Systems [CCITT 80] where a separate computer deals with all signalling information.

When programs are running at different locations to achieve a common, distributed task, the set of rules which defines the dialogue part among the cooperating entities is called a protocol. When looking into a complex distributed system executing an application distributed among several multiprocessor data-processing systems communicating through a packet switched network, protocols are used for the internal operation of the network and for the internal operation of the data-processing systems, as well as for the cooperation of the application programs. In other words this complex distributed system is made of the assembly of several distributed systems in a more complex architecture in which one of the systems is used as a vehicle for the protocols of the other.

This clearly shows that the scope of distributed systems and their protocols is not restricted to user-oriented application but also applies to the intimity of data-processing as well as telecommunication systems. The complexity and the globality of this problem indicates the need for a common reference architecture for all distributed systems as demonstrated in [Zimmermann 83].

In these lectures we first present the OSI Basic reference model [ISO 83a] established by ISO (International Organization for Standardization) joined by CCITT (Comité Consultatif International pour le Téléphone et le Télégraphe) as a structuring technique for distributed systems. We discuss the main concepts used for decomposition and try to show their generality for any distributed system. The OSI model is then used as a guideline for presenting some issues and tools in protocol specification. We conclude our first lecture on overview in presenting some problems which, despite of their commonality to all protocol design, are not very well known and understood: flow control, expedited flow, multiplexing.

The second section - corresponding to the second and third lecture - describes a set of tools developed by the Agence de l'Informatique toward a general Telecommunication Software Factory: Design - Specification - Implementation - Testing of communication protocols.

In the next section we present the OSI Transport protocol [ISO 83b] in its internal mechanisms as well as we give some indication about its formal specification validation and automated implementation using the tools previously presented.

Finally, we conclude with some "protocol games" in order to give the reader some flavour of the kind of problems encountered with protocol design and specification.

9.1. Overview

In this section we first present the OSI basic reference model as a common reference architecture for all distributed systems. Along this presentation we define some keywords and concepts which will be used throughout the other sections.

The second part of this overview emphasizes some of the structuring principles which have been used for building the OSI basic reference model and tries to show their generality and their applicability to any distributed system. The last subsection of this part also gives the current limit of the model and the further studies required.

Finally the last part of this overview enlights some aspects common to all protocol specification and testing.

9.1.1. The OSI Basic reference model

From the complexity of distributed systems it is clear that the construction of systems will be facilitated if all distributed systems would refer to the same decomposition principles, called a common reference architecture. The nature of the problem is so complex that the natural selection of the best architecture among all those experimented by users, suppliers and carriers would take years and years, leading to a situation in which the investments made in divergent experiments would make any convergence impossible. Therefore a voluntary process has been started by ISO and CCITT, resulting in the definition of OSI basic reference model.

External visibility versus internal behaviour

In order not to impose useless constraints to systems, the model defines only the communication part of distributed system, and therefore only deals with the communication protocols and their external visibility (i.e. the behaviour of the system viewed from outside, not its internal organization).

Layering principles

Layering is a structuring technique allowing to view a network of open systems as composed as a succession of layers. Each layer performs a set of specific functions which, in combination with those provided by the lower layers provide a new - enhanced - service to the upper layer. The service offered to the upper layers may differ from the service offered by the layer below either in the nature of the service (i.e. new services are added) or in the quality of the service (i.e. the service is only enhanced) or both.

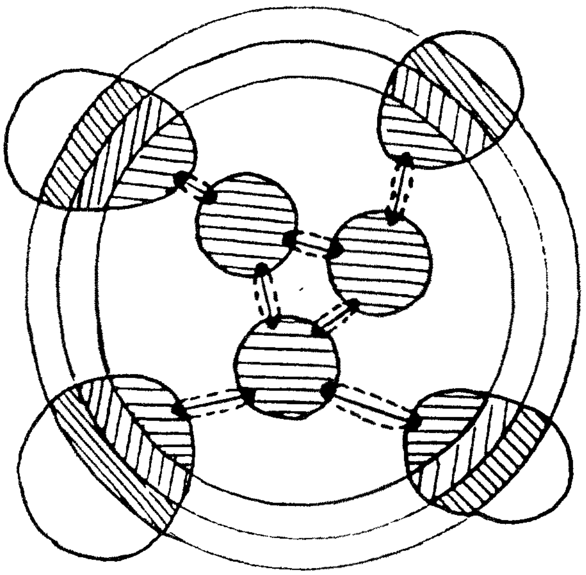


Figure 1: A network of open systems as a succession of layers, each system being viewed as subsystems.

Each individual system is viewed as being an "abstract" open system composed of a succession of subsystems, each corresponding to the intersection of the system with a layer. In other words a layer is viewed as logically composed of all of its subsystems.

Each subsystem is in turn viewed as being made of one or several entities. One entity belongs to only one system and only one layer. All the entities in the same layer are named peer-entities. Since some concepts are layer independent we use the notation "(N)-name" to designate a component or a function which applies to a layer, irrespectively of the actual name of the layer. Application of the above notation leads to the following definition: a (N)-subsystem is the intersection of a (N)-layer with an open system. A (N)-subsystem may contain one or more (N)-entities.

When a (N)-layer communicates with the adjacent layer and higher layer it is also conveniant to use the notation (N+1)- layer and (N-1)-layer.

Objectives of layering – Services – Stability

The goal of layering as a structuring technique is to allow the design of the (N)-protocol to be done in knowing what the (N+1)-layer is expecting and what the (N-1)-layer is providing for, but knowing neither what function the (N+1)-layer is performing nor how the (N-1)-layer is operating. In other words this is to ensure independence between layers. This permits changes to be made in one (set of) layer(s) provided the service offered to the next higher layer remains unchanged. This property is guaranteed if the services provided by the layer are defined independently of how these services are performed.

Communication between the (N)-entities makes exclusive use of the (N-1)-service. In particular direct communication between (N)-entities in the same system is not visible from outside the system and is therefore not covered by the reference model.

The set of rules governing the cooperation between (N)-entities is termed an (N)-protocol: this defines how (N)-entities are working together for offering the (N)-service in using the (N-1)-service and adding their own (N)-functions. The (N)-service is offered to the (N+1)-entities at the (N)-service-access-point or (N)-SAP for short. A (N)-SAP offers service to only one (N+1)-entity and is served by only one (N)-entity, but a (N+1)-entity may use several (N)-SAPs as well as a (N)-entity may serve several (N)-SAPs.

Each layer offers as a common service a way to perform an association between peer SAPs.

The most current association is a bipoint connection between a pair of SAPs. Connection-less-data transmission between SAPs is also now defined as an addendum to the first version of the basic reference model [ISO 82] (multi end-point connection and broadcast are still under study).

For the other concept we encourage the reader to refer directly to [ISO 83a] or [Zimmermann 81].

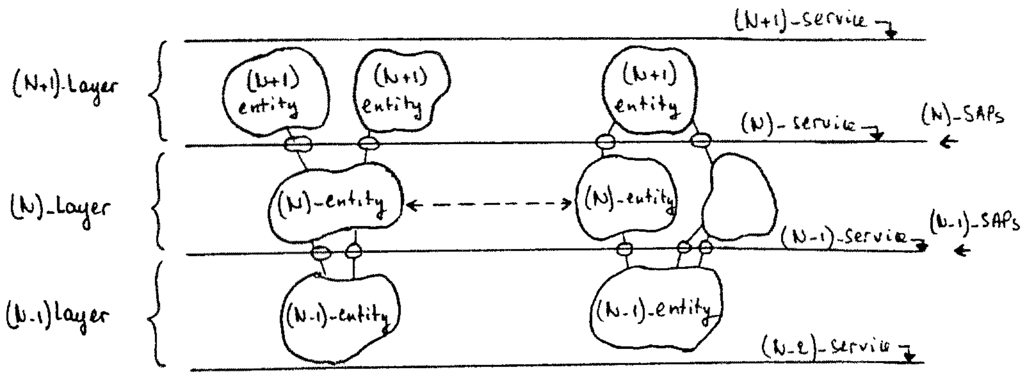


Figure 2: Entities, Services, Protocols, SAPs

The specific layers

The basic model includes seven layers defined as follows:

1. ***Physical layer***. The first and well known function of this layer is to be responsible for interfacing systems to the actual physical media for OSI. The physical layer has also the role of relaying bits when necessary (i.e. performing the function of interconnecting data circuits).
2. ***Data link layer***. The main function of this layer is to perform framing and possibly error dectection and error recovery.
3. ***Network layer***. The network layer relays packets and routes both packets and data circuits.
   Additionally it may perform multiplexing, error recovery and flow control when needed for the optimization of the transmission resources.
4. ***Transport layer***. The transport layer performs end-to-end control and end-to-end optimization of the transport of data between end-systems. The transport layer is the last and highest communication oriented layer which purpose is to hide to the users

the peculiarity of the communication facilities and to optimize their use from the viewpoint of the user. No function related to the transport of information (e.g. recovery from transmission error, multiplexing,...) is allowed to be performed above the transport layer.

5. *Session layer*. The session layer offers common functions used by any dialogue between processes: initialization, different variants of termination, synchronization...
6. *Presentation layer*. The presentation layer offers functions for data formats, code and representation of the informations which application wishes to manipulate: the presentation layer takes care of syntactic aspects of information exchange. Therefore application entities are only concerned with the semantic aspects.
7. *Application layer*. The application-layer performs those functions necessary to achieve a specific distributed task using the services provided by the lower layers.

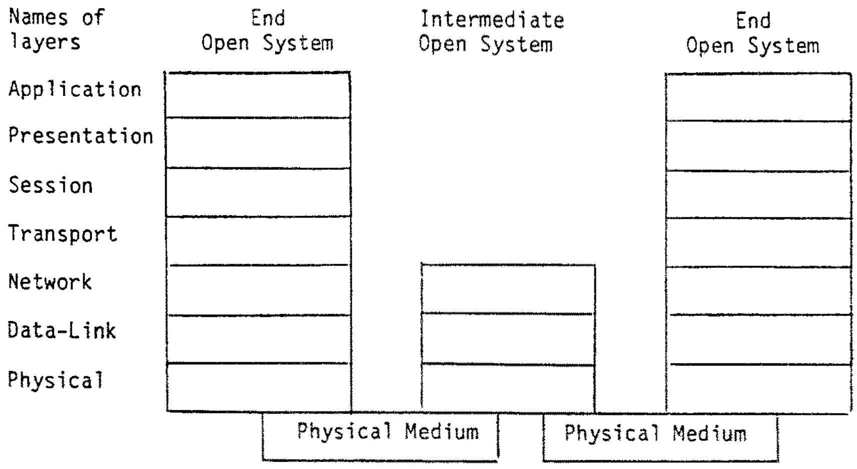


Figure 3: The seven layers of the OSI Reference Model

9.1.2. Basic principles for decomposition

While the previous section presented a brief overview of the OSI Basic Reference Model, this section recalls three basic principles which have been used when building the model and which can be applied for the decomposition of any distributed system [Zimmermann 83].

Separation between data transmission and data processing

The transport service defines a firm boundary between the data-transmission part (layer 1 to 4) and the data-processing part (layer 5 to 7). This is based on the following assumption:

- evolution of technology in both domains should be allowed independently
- the problems to be solved are of different nature:

as an example recovery from error when transmitting is much simpler than recovery in data-processing.

Separation between "end-to-end" and network control

This separation allows to make a clear distinction between

- network control such as routing (finding, changing, reconfiguring routes),
  recovery from line failure or from intermediate mode crash or congestion, controlling flow over network.
- end-to-end transport function managing only simple configuration, based on well known ent-to-end flow control and recovery mechanism.

One could view the network layer as taking into account the interest of the overall community in sharing the telecommunication resources and offering the highest possible availability, while the transport layer offers to its users what they need, on an end-to-end basis without the knowledge of the internal organization of the network - using the network at the lowest cost.

End-to-end transport and network control do not deal with the same kind of optimization, the same kind of resources and therefore do not call on the same kind of functions.

Basic and specific functions in the higher layer

The OSI reference model structures the higher layers (data-processing) as follows:

1. The session and presentation layer contains functions of general use for distributed application. In order to offer this service the session layer [ISO 83d, ISO 83e] contains a set of functions (termed functional units) which can be selected or not on request of the application.
   Those functional units include negociated release, expedited flow, two-way-alternate (TWA) versus two-way-simultaneous (TWS) mode of dialogue, weak synchronization (termed minor marks) or strong synchronization (termed major marks), resynchronization and the possibility to structure a dialogue into phases (termed activities). At session establishment time the application-entities select the functional units they need to achieve their common task. The OSI session sevice and protocol are defined in [ISO 83d, ISO 83e].
   In order to play this role (allow the application entities to share a common view of data without taking care of the syntactic aspects) the presentation layer - which is still under definition - will allow for syntax transfomation and use of predefined types (eg. the structure of a document into pages, windows, graphics,... for a virtual terminal) as well as user-defined types [CCITT 83].
2. The application layer contains the functions which are specific to the application to be executed.

It is clear that this structure has been largely influenced by high level languages (as well as by operating system concepts). The reader will find in [Zimmermann 83] a comparison between some functions in OSI and in traditional data processing.

Despite of the fact that some further study is still necessary both in the session layer (multiparty dialogue, commitments protocol,...) and the presentation layer, we share the view expressed by H. Zimmermann that any distributed system architecture should adhere to the same structure as OSI for the higher layers.

9.1.3. Some basic concepts and terminology for protocol design and specification.

This section introduces some basic concepts and terms as defined in [ISO 83a] which knowledge and understanding is useful for protocol design and specification.

Identifiers

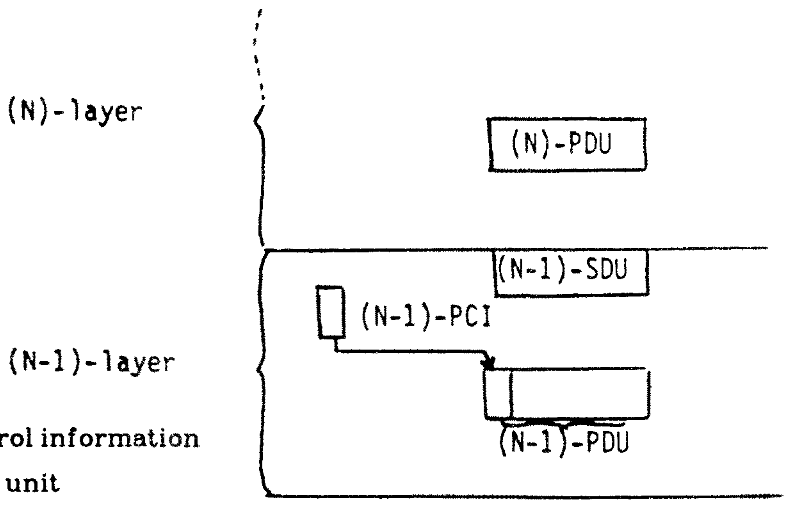
Objects within a layer or at the boundary to the layer above and below need to be identified.

- Each (N)-entity is uniquely identified by a global title in the network of open systems.
- Within a specified domain a local title identifies the (N)-entity in the domain. A domain can be defined geographically or for one layer or any other combinations.
- A (N)-SAP is identified by a (N)-address at the boundary to the (N+1)-layer.
- A directory allows to find the (N-1) address through which a (N)-entity having a given global title can be reached.
- A mapping function gives the correspondence between the (N)-addresses served by a (N)-entity and the (N-1)-address(es) used for this purpose. Address mapping may be done by table or on a hierarchical basis (in which case the (N)-address is formed or the (N-1)-address completed by a (N)-suffix).
- A (N)-CEP (connection end-point identifier) is used to distinguish inside a SAP between different connections.

Data-units

- (N)-SDU (service data unit) is the amount of data whose integrity is preserved from one end of a (N)-connection to the other. In some services the length of the SDU is unlimited (e.g. X25).

- (N)–PDU is an unit of data exchanged between two (N)–entities, using a (N–1)–connection, when operating the (N)–protocol. A (N)–PDU contains (N)–protocol control information and possibly (N)–user data (which is a (N)–SDU or a part of a (N)–SDU or serveral (N)–SDUs. In X25 the network–protocol–data–unit is the packet. A data packet contains either a complete network–service–data–unit or a part of a network–service–data unit. The 'more data' bit allows for the preservation of the integrity of the NSDU



PCI: Protocol control information
PDU: Protocol data unit
SDU: Service data unit

Figure 4: Logical relationship between data units in adjacent layers.

Relationship between (N) and (N–1)–connections

The operation of a (N)–connection requires the use of (N–1)–connection(s).

Three types of combinations are of particular interest:

- One–to–one correspondence where each (N)–connection uses one (N–1)–connection, and one (N–1)–connection is used for only one (N)–connection.

- Multiplexing where several (N)-connections use the same (N-1)-connection.
- Splitting where one (N)-connection uses several (N-1)-connections. (Note that splitting and multiplexing can be combined).

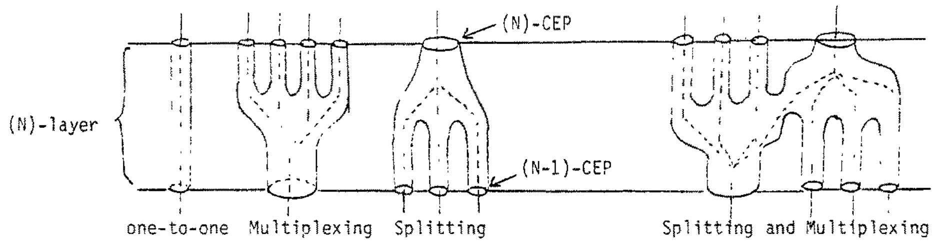


Figure 5: Correspondence between connections

Flow control

Some services offer a flow control service allowing the receiving user to slowdown the rate at which the service delivers to it the SDUs. The service propagates (if necessary) this regulation to the sender.
This flow control service may be offered at the protocol level using different techniques which can be classified in two main categories:

- explicit techniques where the protocol includes its own mechanism for flow regulation.
- implicit techniques where the protocol makes use of the flow control provided by the lower service. In this case there is a risk of long term blocking which may lead to an unacceptable degradation of the service.

Expedited data

Some services offer an 'expedited data' transfer service allowing for the transmission of a short piece of 'fast' data. An expedited data may bypass normal data, in particular when the receiving user exercices flow control on normal data. Experience using expedited data has shown that this is a powerfull service - in particular for forcing resynchronization when the receiving user is not willing to accept normal data any more. However unexpected bypass of normal data by an expedited is a source of errors in protocol design (e.g. an expedited carring a resynchronization request arrives before the normal data carring the synchronization mark referred to in the resynchronization request).

9.2. Toward a telecommunication software factory

This section presents a set of integrated tools developed by the Agence de l'Informatique in France for helping the protocol designers, implementors and users at each stage of the 'life' of a protocol.

At the first stage the protocol is designed by users, suppliers or standardization committees. During this stage the protocol should be specified and validated.

At the second stage the protocol will be implemented, possibily by different teams, in different systems.

At the last stage, the protocol will be put into operation in real networks with possible maintenance of the equipments running the protocol (or of the protocol itself !).

At first glance it appears that specification - validation are tightly coupled. However our methodology proposes a refinement of this stage into two steps:

- The first step includes an initial specification of the protocol in which some details may not be included (like encoding, mapping of (N)-PDU into (N-1)-SDU...) and a validation of this specification. The tool for helping during this stage is called LISE and is presented in the first subsection.
- The second step consists in a complete specification of the protocol (including those details which have been missed in the first step) using a formal specification language called PDIL. This also includes simulation, study of properties as well as automated

implementation.

From one hand this provides a reference description which can be used by any implementor, from another hand this builds all the environment for conducting real experiments at low cost and provides help for a smooth transition toward easy implementation. PDIL is presented in the next subsection.

Whatever the tools are for helping the implementors –including an automatic translation from the specification into machine executable code – it is likely that a lot of implementors will implement protocols manually in assembler language because of specific performance or environment constraints. Therefore this raises the question of testing equipments which claim to be in conformance with the protocol. The third subsection presents some testing tools which can also be used when the network is operational (i.e. during the third stage).

9.2.1. LISE

LISE is a tool based on extended finite state automata (state automata with predicates). This form has been chosen since it is the more popular in the community of protocol designers (i.e. suppliers, carriers, standardization committes).

The concepts

According to the OSI Reference Model a (N)–protocol is run by peer (N)–entities – 2 or more – using an–(N–1)–service and providing a (N)–service. The behaviour of each (N)–entity is described using an extended automata, by a set of transitions of the following form:

```
<trans>:: = '('<event> <fromstate> <predicate> <tostate> <action> ')'
<event>:: = nothing |<input–event>
```

Nothing means that this is a spontaneous transition which can fire at any time provided the state and predicate are as specified in the other part of the transition.

<input event>:: =	N-service request or response
	|(N-1)-service indication or confirmation
	|(N)-PDU
<from state>:: =	state-name
<to state>:: =	state-name
<predicate>:: =	a boolean expression calculated using parameters of the event and variables. The variables are in fact an extension of the state of the process.
<action>:: =	<action-on-variable> <action-to-(N-1)-service> <action-to-(N)-service>
<action-on-variable> :: =	nothing | set-variables
<action-to-(N-1)-service>::=	nothing | send (N)-PDU send | an (N-1)-service request or response.
<action-to-(N)-service>:: =	nothing | send (N)-service indication or confirmation.

Then a model of the (N-1)-service is introduced and the (N)-entities are interconnected through this model.

We have gained from our experience that constructing the model of the (N-1)-service may be more costly than building the model of the protocol itself.

For this reason LISE offers a set of predefined models corresponding to almost all existing networks. The predefined models are listed in the section on "(N-1)-service models", page 498.

After the model has been selected (or constructed by user if none of the preexisting models fit into the (N-1)-service which is to deal with) the properties of the overall communication can be studied. This study may include: validation through global state exploration, simulation, study of properties.

User interface

LISE has been designed to be an interactive tool, therefore it includes an ease-of-use user interface which essential features are:

- A transition oriented editor including functions such as searching or

deleting transitions using criteria like:
list all transitions including a specific object as a component ... etc.

- Save–restore a set of transitions into/from a specified file.
- Check the properties of a local state automata: connexity, paths, cycles, sink state...
- List the objects (i.e. the components of the transitions) and their characteristics and check them for consistency (for instance a state and a (N)–PDU shall not have the same name, or a (N)–PDU which never appears as the event of a transition cannot be a (N)–PDU!).

The user interface also includes some facilities which are called on after a validation has been done – they are presented later.

Finally it should be noted that LISE is a bilingual system which proposes to its user to use either French or English.

Validation

We will first consider the case of two (N)–entities and then describe the extension for n. Similarily we first consider the case of a simple (N–1)–service comprising two fifo queues without flow control (i.e. when the queue is full this is an error).

The method is very simple and based on a global state exploration. As proposed in [Zafiropoulo 80], starting from an initial global state in which both processes are in their initial state and the channels are empty, the reachability of the system is build in studying all possible transitions done by the two processes. A data base contains all the global states and when a new global state is computed it is first checked against the already exiting global states in the data base and added to the database only if it does not already exist. The validation stops when the tree has been globally explored (i.e. no new global state can be created).

The method allows the detection of three kinds of errors:

- unspecified reception: Reception of an event ev, head channel P_i ––> P_j, when the process P_j is in a state S such that there are no transitions

<ev S predicate ...> for which the predicate is true.

• deadlock:	The global state of the system is such that no transition can be executed further.
• non executable transition:	At the end of the communication analysis a transition has never executed.

In case of errors the system may display upon request of the user a 'history' of the error.

In parallel with the construction of the reachability tree of the communication, the global state automata is constructed (i.e. not only the global states but also all the transitions between these states are put into the data base) and a new part of the user interface is then available:

- Display all or part of the global state automata.
- Study its connexity, cycles ...(note that the deadlock detected by this method has less power than the liveness property in petri-nets, which has to be studied through the connexity of the global state automata).
- Display communication scenarios: a global covering of the state automata is constructed and then the corresponding transition sequences are displayed. This can be further used either for protocol teaching purpose (a special extension of LISE, called a protocol teacher is also available for this purpose) or as test suites when performing equipment testing.

Simulation and properties study

While in validation mode the system fires all possible transitions in every global state, when turning to simulation mode the system selects only one transition. This selection is done on a random basis. This mode is useful when it turns out that a validation cannot be run due to a too high number of global states.

An other feature allows to put low priority to transitions corresponding to error cases (for instance). Therefore when running a simulation, the error cases are considered with a lower probability than the normal operation cases.

It is very well known that absence of deadlock does not proof that the protocol operates properly: it is very simple to build a (so-called) mutual exclusion algorithm which does not fall into deadlock but allocates the resource more than once simultaneously. One key drawback of the state exploration method is that it provides no tool for verifying that the protocol meets certain requirements. Such a feature has been added in LISE in the form of global assertions.

A global assertion is nothing but a spontaneous transition whose predicate can check any component of the communication (including the remote process, and the (N-1)-service). In fact the predicate of such a transition is true when the assertion is false and then the process goes into an error state. Global assertions have the highest priority in order to be still detected in simulation mode.

(N-1)-service models

The basic (N-1)-service model consists of two fifo queues. LISE proposes to the user to build its (N-1)-service model in adding to this basic model any combination (except certain combinations which are senseless like datagram and flow control or datagram and expedited data) of the following properties:

• purge:	The fifo queue can be purged on request, each direction independently.
• complete purge:	The fifo queue can be purged on request, simultaneously, including a purge collision resolution algorithm (like the X25 reset).
• flow control:	The receiving process can block/unblock the channel and the channel can block/unblock the sending process.
• expedited data:	Each element put in the queue can be characterized as normal or expedited. Expedited elements may bypass normal elements. All combinations resulting from bypassing are

considered by the system.

- datagram: The order in which the elements are delivered is independent of the order they have ben put into the queue (misodering).
- datagram with loss: Similar to the previous one except last that the (N-1)-service may also loose any data.

The properties selected by the user are represented in the form of parameters which are used by the validation/simulation algorithm when building the next global state(s). The system also offers global variables which can be used to add user-defined properties to the (N-1)-service. These global variables can be checked/set in every transition.

Timers

As an important feature, the system includes a timer management facility. A property is added to the channels which is to define a minimum transit delay and a maximum transit delay for the elements put into the channel.

An actual transit delay is attached to each element in the channel and initialized to zero when a process puts an element into a channel. When exploring the global states the validation algorithm

- does not deliver an element if its actual transit delay is less than the minimum.
- forces the delivery of the element if its actual transit delay has reached the maximum.
- progresses the time if no transition corresponding to a timer which has run out or to the delivery of an element having reached the maximum transit delay can be fired.

  "Progressing the time" means:
  - increment by one the actual transit delay of every element in the channels
  - execute a user-defined action (like decrementing a counter modeling a timer which has been started).

Unlike classical simulation systems there is no 'virtual clock' in the sytem (such a variable would prevent the validation algorithm to ever terminate).

Extension to n processes

Although very powerful, the main drawback of the system is to be limited to the study of the communication between two processes. An extension allowing to interconnect n processes through a (N-1)-service has been recently developed and is currently under test.

While figure 6 gives the general configurations of LISE in the 2-process version, figure 7 gives the configuration of the extension in the n-process version.

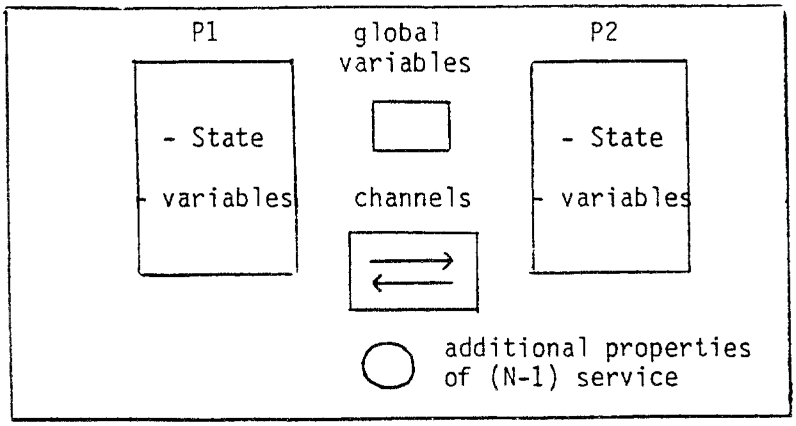


Figure 6: LISE in the 2-processes version (model)

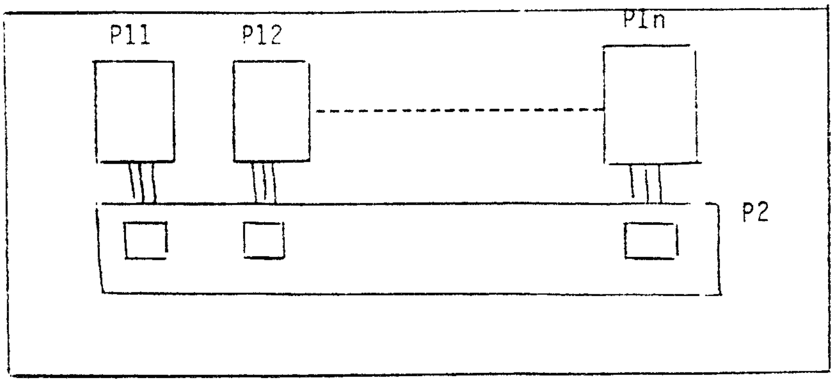


Figure 7: LISE in the n-processes version (model)

In the n-processes version there are still two "processes" P1 and P2.

- P1 is declared as having n instances P11... P1n representing the n (N)-entities to be interconnected. Each instance has its own state and variables.
- Each instance of P1 communicate with P2 through an interface modelled by two channels which are fifo queues P2 has a global state and context-states (one context is attached to each channel). P2 is used to model the (N-1)-service. The validation/simulation algorithm again explores all transitions and builds the global states.

  A global state is a vector comprising

  - the state and variables of P11...P1n
  - the state and variables of P2
  - the context-state and context-variables of all contexts of P2
  - the global variables
  - the content of the interface channels.

It is clear that the number of global state is largely growing with this configuration. Therefore if validation cannot be used, the system automatically turns to simulation mode.

Conclusion

The LISE system is operational under Multics on a Honeywell 68 large main frame computer. It has been used for validating example protocols (HDLC/X25) but also real protocols during their design phase ISO Transport, ISO Session, file transfer.

The format of the input (transistions) facilitates the setting up of the system since it is close to what is used by standardization committees. The global assertion feature has been felt of primary importance when validating the transport protocol [Ansart 80].

The possibility to easy select a complex (N-1)-service model has been largely used when validating a subset of the OSI session protocol [Ansart 83a] and has permitted to conduct a significant validation of this complex protocol in a short time [Ansart 82a, Ansart 82b].

The reader familiar with the french language may find in [Ansart 83b] a complete user's manual of LISE.

9.2.2. PDIL

PDIL (Protocol description and implementation language) is a language developed by the Agence de l'Informatique – and also set of associated tools: a compiler (or more appropriately a preprocessor) which translates a PDIL program into a PASCAL program, a simulator which executes the PASCAL programs produced by the preprocessor and finally a set of run-time environments which allow to integrate the PASCAL programs into real operating systems and execute them as an automatic implementation of a protocol.

Concepts

PDIL allows the description of both services and protocols. For describing a protocol the technique used in PDIL is to describe a (N)-entity running the (N)-protocol. When specifying a service, the corresponding PDIL program describes it as a box. The main difference between these two units of description is that in the case of an entity there are service access point(s) at which the entity is offering service and service point(s) at which the entity is using services, while a service unit of description does not use other services.

In general, a service is not subject to automatic implementation but is only used for simulation purpose. This is the case when we describe a distributed service in the OSI sense (e.g. the transport service). However local services may also be described and therefore implemented (e.g. a memory management service). We will focus on entity description in the remainder of these lectures.

Model and Instances. The unit of description in PDIL is in fact a model of an entity which represents all possible behaviours of a (N)-entity respecting a (N)-protocol. At implementation time a system will support one or more instances of this model, each instance being derived from the model in fixing parameters. The behaviour of each instance conforms to the model.

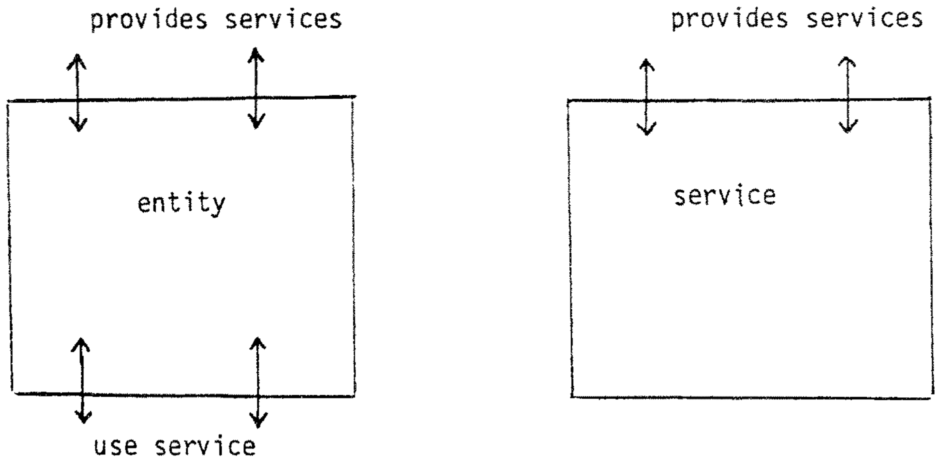


Figure 8: Entity and services unit of description

Parametrization. Four levels of parametrization are offered in the PDIL language in the following way: constants, types, variables and procedures/functions may be declared as "external" with the following semantic: whatever their values are, the behaviour of any instance giving to them the actual values conforms to the behaviour of the model which specifies the protocol.

Machines, channels, contexts. Entities in a system are communicating through channels. Channels are bidirectional fifo queues which can be dynamically allocated and destroyed. Additional intelligence can be attached to channels (for instance resolutions of collision when closing a channel).

An entity can be splitted into several machines. As entities, machines communicate through channels (no shared memory).

An (N)-entity does not need to know the internal structure of the (N-1) and (N+1)-entities (i.e. how they are splitted into machines).

Inside a machine, several contexts can be dynamically created/ destroyed. Therefore, a machine has the capability to multiplex several activities in parallel.

State automata. The behaviour of a machine is described in terms of an extended finite state machine as follows:

- The default context of the machine contains a major state and variables (minor states) for the overall machine. This is used for controlling the general behaviour of the machine. In general only a few states (like not-operational, operational, shutdown-in-progress) are used. Interactions asking for creation of new activities are processed in the default context (e.g. a new connection is rejected in the shutdown-in-progress state).
- Each created context contains also a state and variables. When interactions dealing with an existing activity are received through a channel they are directed to the appropriate context in which they are processed according to the state automata of the context.
  Channels are dynamically attached to context in order to have this association done automatically for all interactions received via channel.

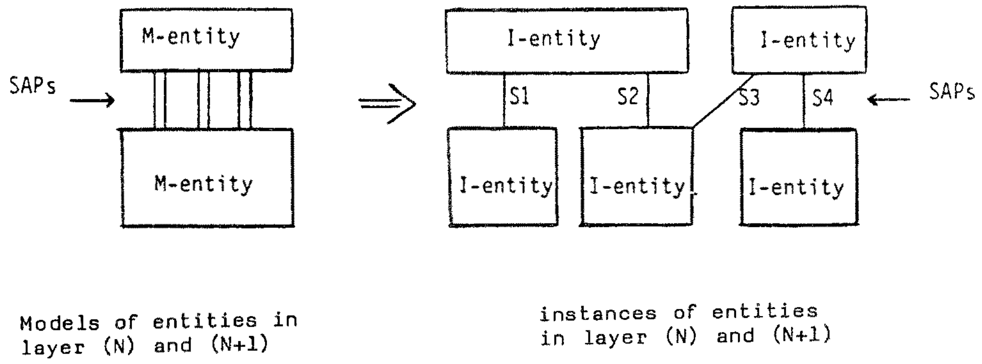

M-entity
M-entity
SAPs
I-entity
I-entity
S1
S2
S3
S4
SAPs
I-entity
I-entity
I-entity
Models of entities in layer (N) and (N+1)
instances of entities in layer (N) and (N+1)

Figure 9: Use of SAPs for addressing instances of entities.

Addresses and service access points. Entities are connected at service-access-points (SAPs). Each SAP is uniquely identified by an address.

When a channel is created the address of the SAP is given as a parameter. This allows several instances to be created in adjacent layers at implementation time, in conformity with the model.

Moreover this allows for separate compilation and implementation of the layers: adding new users only requires to change the SAP table in the system.

Language features

The language used is an extension of Pascal. Pascal has been chosen due to its popularity, good structuration, typing facilities and the fact that it is standardized.

A unit of description starts with a header of the following form:

entity name1 ***providing-to*** name2 ***using*** (name31,... name3n);

name1 defines the service which is provided. There could be several descriptions providing the same service. Each instance will use one of those at implementation time.
name2 is only used internally to designate the potential users of the service. This header is used to provide linking information between separate descriptions of entities and services, therefore allowing for separate compilations.
name3i designates other services which are used by the current description. They shall be defined somewhere else as 'name1' of other description.

As is Pascal, labels, constants, and types are then introduced, with the difference that constants and types may be 'external' (i.e. not defined).

The next section gives the components of the entity, mainly:

- the names of the machines composing the entity,
- the name of the channels to be used,
- the structure for the addresses of the SAPs,
- the name of the interactions to be exchanged through the channels,
- the structure of the interactions in the form of Pascal records,
- the structure of the contexts and the state space of each of the state variables.

Then the behaviour of the entity is specified in the form of an extended state machine, through constructs of the following form:

```
when interaction-name
    from state-name
        provided predicate
            to state-name
                .
                .     action
        provided other-predicate
            .
            .
    from other-state-name
when interaction-name
    .
    .
    .
```

Undeterminism

One important feature of the PDIL language is that it allows to describe an undeterministic behaviour. This is of prime importance when describing protocols in order to avoid over-specification. Undeterminism can be introduced

- in calling on external functions which return a result having a specified type but an undefined value,
- when more than one predicate (provided clause) is true and one of the actions corresponding to one of those predicates may be executed.
- Spontaneous transitions may also be introduced: they are transitions which are not triggered by any external event. A spontaneous transition can fire at any time, provided a set of conditions based on the internal state of the entity is true.

Other facilities

A state automata description implies that there are well identified events which are received in well known states. The problem in protocol description is that identifying the event itself is a part of the protocol description. Similarily identifying to which state automata instance the event applies is also a part of the protocol.

When a (N−1)−SDU is received, it is first necessary to recognize a (N)−PDU (i.e. identify the event), then find to what connection the PDU belongs (i.e. select the state automata instance) before applying the transition of the state automata.

PDIL contains all the appropriate features for describing this part of the protocol.

When a (N−1)−SDU is received it is first processed in the context attached to the channel to which the interaction carrying the (N−1)−SDU has been received.

Then a special decoding function is called, which allows for recognizing the PDU and selecting the appropriate event.

If necessary, the parameters of the PDU may be used to identify what automaton's instance the PDU belongs to.

Then the appropriate context is selected before the state automata is called. Context can be selected by identifier or by criteria. Other constructs allow to apply one event to several contexts in turn (e.g. all contexts meeting a particular criteria).

Abstract memory management

Additionally, PDIL includes a facility called 'abstract memory management' allowing to describe what happens to the data passed by the (N+1)−entity and the (N−1)entity. A set of system calls (fragment, assemble, copy, forget, create, expand) are offered to the user for describing user−data manipulation. This approach offers tools for an unambiguous description of this important aspect of a communication protocol and has permitted to achieve automatic − and efficient − implementation.

Automatic implementation and simulation

Automatic implementation and simulation both rely on a compiler (a preprocessor) which translates PDIL source into PASCAL programs.

The preprocessor. The preprocessor - operational under Multics - translates a PDIL description into

- a main Pascal program containing one procedure per interaction belonging to the entity.
- a set of Pascal subroutines which may be either complete (i.e. they contain a completely programmed body) or to be completed by the user at impletation time.
- a set of tables containing internal information on the structure (machines, interactions, ...).

The PDIL preprocessor makes the syntax checking and produces a Pascal code independent from the target system.

Automatic implementation. Automatic implementation is the creation of instances corresponding to the model previously described. The Pascal programs produced by the preprocessor should be completed (in order to become executable):

1. with a run time environment offering the system calls used by a PDIL description: mainly abstract memory management and channel management.
2. with some part of the protocol which may not be fully specified (e.g. detailled encoding/decoding).
3. with the parameters and subroutines characterizing the instance.

The run time environment differs depending on the target system on which the automatic implementation is to be executed. But it has to be constructed only once per target system and can then be reused for any protocol implementation in any layer on the same system.

The part 2 referred to above is specific of each protocol and should be coded manually for each new protocol which is implemented.

The part 3 may be reduced to some simple tables or may be comprise some sophisticated subprograms depending on the protocol (like an algorithm to optimize local resources).

Our experience has shown that, when the run time environment has been constructed for a given operating system, more than 90% of the code representing an implementation is obtained automatically.

The simulator. The simulator is nothing but a specific run time environment which executes the code in a controlled manner. Simulation of the (N)-protocol requires that the (N-1)-service has also been specified. A typical configuration of a simulation is given in figure 10.

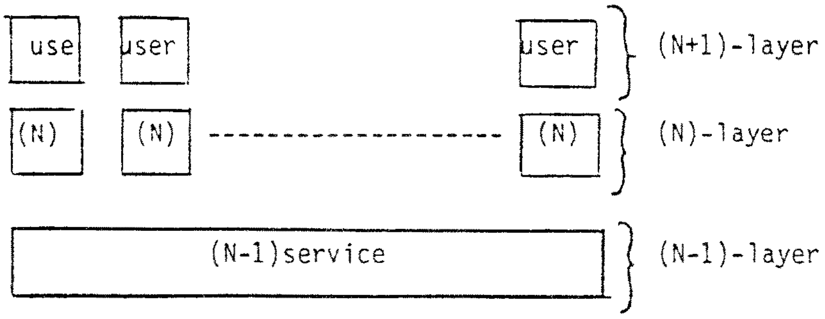


Figure 10: Typical configuration of an (N)-protocol simulation

As can be seen in figure 10, the simulation also requires users in layer (N+1). The users excercise the service provided by the (N)-entities. The users are also written in PDIL as very simple (N+1)-entities. They include spontaneous transitions representing all possible behaviours of the user of a the (N)-service.

Conclusion

From experience it appears that PDIL and the associate tools are really the basis of a protocol software factory. The preprocessor, the simulator and a run time environment on INTEL RMX 86 are operational. A lot of protocols in layer 2,3,4 and 5 have been specified using PDIL [Ansart 83c] and the experiments of the automatic implementation on RMX86 are very promising – more than 90% of the code is produced automatically and the mean time for implementing a protocol is reduced by more than 70% compared with traditional implementation methods for protocols. New experiments are in progress using UNIX.

PDIL is close to the work developed at the Montreal University by the team of G.V. Bochmann [Bochmann 82], at NBS [Blummer 82] and also to the language currently in development by ISO/TC97/SC16 [ISO 83f].

Overview of PDIL in English can be found in [Ansart 83d] and the reader familiar with the french language will find in [Chari 83] a complete specification and user's manual of PDIL.

9.2.3. Testing tools

As soon as a distributed system involves several components built by different parties - e.g. users, suppliers, ... - and communicating using protocols, setting up the system does not end with the protocol design and implementation steps: verifying that the components conform to the protocols - i.e. that the equipments really respect the protocols when running is also an important task.

Purpose of testing and testing tools

First, testing tools should provide help for debugging implementations while building the system. This includes mainly two aspects:

- checking that the equipment correctly run the protocol;
- checking the robustness of the software against abnormal situations: reaction to protocol errors made by another party as well as to transmission errors (i.e. errors signalled by the (N-1)-service).

Secondly, testing tools should provide help for performing acceptance testing: when a component of a network is installed by its user who may have bought it from a supplier, the user should be able to verify that:

- the equipment conforms to the protocol,
- the range of options really supported conforms to what is claimed to be supported by the supplier,
- the equipment has an acceptable degree of robustness,

- the performance meets the user requirements.

Finally, after equipments have been put into an operational network, the experience has shown that there is a need for:

- arbitration facilities: in the case of abnormal behaviour it is of prime importance to be able to designate without ambiquity the faulting equipment;
- online measurement facilities: this covers traffic measurement (in order to anticipate the possible evolution of the network topology) – as well as specific protocol performance measurement (like the ratio between the number of data messages versus control messages) in order to prepare the next versions of the protocols themselves.

Specific constraints for a testing tool in an open environment

For systems under test which have implemented an interface permitting direct access to a specific layer, it will be possible to test this layer, provided that the lower layers have already been tested.

If this is not possible, combined testing of layers will be necessary.

For example, once the transport layer has been tested in this way, the session layer may be tested. Then, after this, it will be possible to test a virtual terminal protocol or a file transfer protocol. Conversely, if the transport layer interface is not available, but the session interface is available, the transport and session layer will be tested together, before the VTP is subsequently tested.

This flexibility is of prime importance, because a system will make the choice not to exhibit a particular interface for performance or architectural consideration e.g. an X25 chip does not provide external access to the layer 2 service interface.

Another fundamental constraint for a testing tool is its ability to accommodate different versions of protocols: if the cost for building the testing tool when changing the protocol is higher than the equipment production cost, then the testing tools will never be used... This aspect is very important when several versions of a protocol are experimented. This property is sometimes referred to as 'protocol independence' of the testing tools [Ansart 81].

Brief overview of some testing tools

In parallel with the development of the OSI protocol, some countries have decided to start an extensive study of testing tools for equipment implementing the OSI protocols. In the USA, the NBS has set up a set of tools for the OSI higher layer (layer 4 to 7) which are now operational for the transport protocol [Nightingale 81, Nightingale 82].

In Germany the GMD concentrated on teletex layer 4 [Faltin 83] . In the UK, the team leaded by D. Rayner at NPL has developed testing facilities for the network layer over X25 [Rayner 82], while in France, the Agence de l'Informatique received the task to study tools for layer 4,5 and 6.

This subsection briefly presents three testing tools developed by the Agence de l'Informatique in France [Ansart 83e, Damidau 82, Bonhomme 83, Ferret 83].

The STQ system. The STQ system is based on two main components

- the reference system,
- a distributed application which uses (and exercices) the service provided by the reference system and the implementation under test.

The distributed application contains mainly two parts:

- a test driver which is the active part executing a scenario,
- a test responder acting as a passive system which responds to the stimuli sent by the test driver via the (N)-service. The test driver and the test responder are cooperating in executing a scenario.

A scenario is a set of commands requiring (N)-services. The test driver transmits to the test responder the commands to be executed in (N)-SDUS of a previously opened (N)-connection. To some extent the responder may be viewed as an interpreter which is remotely loaded by the active tester. Execution of a scenario is splitted into several phases, each of which starting with the transmission of the commands to be executed in the second part of the phase. Since the responder is a quite complicated program, it is important to reduce the cost of its production, as well as to ensure its correctness. In order to achieve these two goals the test responder is splitted into two parts:

- One is the main program comprising the algorithm for command loading, interpretation and execution. This part is written in Pascal and portable. It is given to each potential client of the STQ test system. This part assumes a (N)-service strictly identical to the theoretical (N)-service defined by ISO for the (N)-layer under test.
- The second part deals with the mapping of the (N)-service abstract interface into the real interface available inside the implementation under test. This part is system dependent and shall be realized by the user of the implementation under test. (I..U.T).

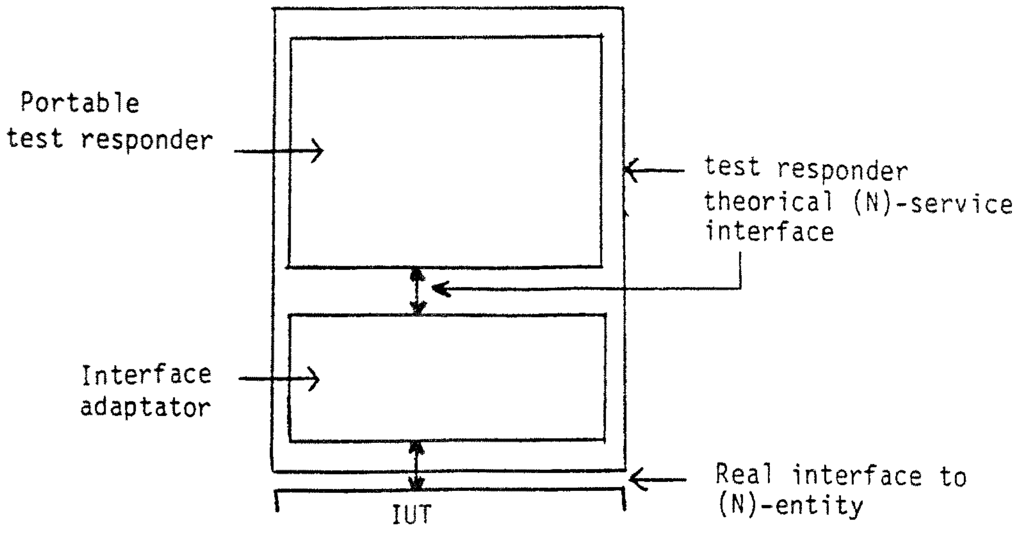

Portable test responder
test responder theorical (N)-service interface
Interface adaptator
Real interface to (N)-entity
IUT

Figure 11: The test responder.

As shown in figure 12, the STQ system contains the following components:

- the reference implementation runs the (N)-protocol;
- the management and configuration module allows for tracing the events, configuring the test (setting parameters and options) and also introducing errors and simulating abnormal behaviour;

- the test driver controls the execution of the tests and pilots the remote test responder(s);
- the (N−1)-service implementation runs protocols (N−1) ... to 1 in order to provide the (N−1)-service used by the (N)-entity.

The scenario commands are divided into three subcategories:

- the supervision commands allow the user of the systems (i.e. the operator of the STQ test system) to set up the configuration, and activate/deactivate the trace and log facilities;
- the operator commands for building/modifying, loading/executing, starting/suspending/resuming scenarios;
  automatic as well step by step mode of operation are available;
- the scenario elements themselves: operation of the (N)-service, remote loading of the test responder, and execution of a distributed scenario (send/receive data, synchronize, echo, ...).

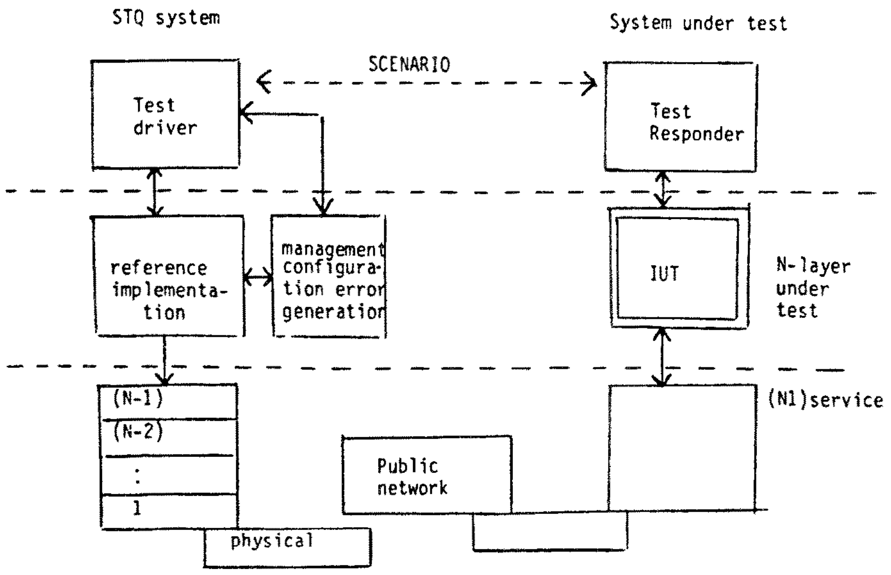


Figure 12: The components of a STQ system.

A traductor based on an syntax analyser translates the external form of a scenario (i.e. a form suitable for the operator) into an internal form (i.e. a form suitable for execution by interpretation).

In order to perform tests starting from elementary tests and increasing the complexity and the completeness, a scenario data base is also available. The tests recorded in the data base fall into four categories: elementary tests for service availability, tests requiring configuration, tests of protocol options and parameters, tests including error recovery and reaction to abnormal situation (error generation). Mono–connection as well as multi–connection tests can be also be run.

The CERBERE tool. CERBERE is a tool designed to be introduced between two equipments running high level protocols. One equipment is the implementation under test, the other one is the reference equipment (i.e. a testing center or an equipment previously tested). The CERBERE cannot provide directly for a complete test system but is only a complementary aid. CERBERE is also useful for arbitration and measurements in an operational network.

CERBERE acts as a relay in layer 3, allowing for

- analysing the user of the network service made by the higher level protocols,
- simulating errors in the network service.

CERBERE offers to its user an interface allowing for decoding/analysing the high level protocols contained in the network service data units. In parallel mode the analysis is done simultaneously with the relay function therefore guaranteeing that the traffic is not perturbed. In serial mode the analysis is performed before the relay function is called on, allowing for 'intelligent' perturbation of the network service (e.g. simulate a network disconnect when the high level commit protocol is in its 'window of vulnerability').

CERBERE also offers a lot of complementary services like: statistic calculation tools, network usage cost computation, sophisticated full screen display of results, traffic storage on disk for deferred off-line analysis ... etc.

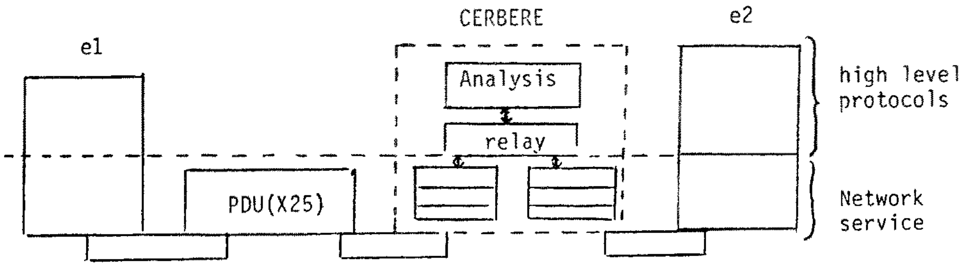


Figure 13: CERBERE as a high level protocol analyser

The GENEPI tool. GENEPI is a protocol data units generator which is indepedent of the protocol: i.e. changing the protocol supported by GENEPI is an easy (and costless) task.

- GENEPI is designed to manipulate conceptual objects involved into a (N)-protocol: (N)-PDUs, access to (N-1)-service, mapping of PDUs into (N-1)-SDUs, encoding/decoding, ...
- The basic software knows neither the format of the PDUs (the syntax) nor of tables and automata that the GENEPI basic software uses to generate the (N)-protocol.

Assuming that GENEPI is designed to generate (N)-PDUs, its basic software contains:

- an implementation of the (N-1)-service
- a set of commands to access the (N-1)-service
- a set of commands to manipulate local variables
- a set of commands to compose and send (N)-PDUs and to decode (access to the fields of) received (N)-PDUs
- a facility for macro-commands
- a state automata-driver
- trace, logging and display functions

In the first stage, the operator introduces the format of the PDUs in two different ways: the logical format describes the PDU's fields as "records" while the physical format deals with the actual PDU's bit strings and the mapping between logical and physical formats.

GENEPI can be run in manual mode: the operator composes logical PDUs with the help of the system (prompting for each parameter), then the system encodes and sends them. When receiving, the system decodes the arriving PDUs und displays them in logical format.

Parameter values can be set to/assigned from local variables for speeding up the protocol operation. This mode allows to set up elementary – but significant – tests of high level protocols in a very few days.

Another mode – termed automatic – is also available: the manual operator is replaced by a set of interconnected state automata, whose events are (N)–PDUs and (N–1)–service indications, and actions are GENEPI commands. Automatic mode permits to build an acceptance test which can be run without the help of any operator. The system includes multiple state automata instances selection/manipulation, therefore complex functions like multiplexing – splitting – error recovery are easy to introduce into GENEPI.

Finally, the system provides for a multilayer testing facility in which PDUs of two adjacent levels (N and (N+1)) can be manipulated simultaneously. The GENEPI package has the network and transport service implemented as standard (N–1)–service: it can therefore be used for testing any protocol in layer 4,5,6 of the OSI architecture. It has been used to test the early implementations of OSI made by French suppliers and PTT.

Conclusion

Testing the conformity of equipments to protocol is one of the key points for the OSI protocols development and use: the objective of OSI will not be achieved unless products are produced in conformance with OSI and tested for this conformance. Although some early testing tools are available – with promising results, there is one main area which has not really been tackled so far: how can the specification tools automatically produce the testing tools:

- deriving the test scenarios automatically from the specification,
- implementing the testing center automatically from the formal specification of the protocol,
- generating automatically analysis programs to be run in an 'observer' (like CERBERE).

Although a tool like LISE produces test scenarios as a result of the validation process, the number of scenarios is too high for being used in practical tests and there is no tool for selecting an useable – and significant – subset offering an appropriate test covering.

Although implementations can be automatically produced from a formal specification, this does not address the problem of error generation and does not help for producing the test scenarios.
The only area in which significant results have been obtained so far is in deriving an automatic observer for the specification of a protocol [Ayache 79].

9.3. Example: The OSI transport protocol

The transport layer is the last layer of the communication oriented layers of the OSI reference model.

Its purpose is to isolate the processing oriented layers from the variation of the quality of service of the network service. It also allows for the optimization the use of the network service on an end-to-end basis.

9.3.1. The transport service

The transport layer provides the transport service [ISO 83c] by adding to the network service the functions supported by the transport protocol.

The transport service offers point-to-point transport connection between transport-service-access points. More than one transport connection may be opened between the same pair of T-SAPs: according to the model they are locally distinguished by means of Transport-connection-end-point identifiers (T-CEP-ID).

A transport connection comprises three phases:

- The connection phase allows for opening the connection between a pair of specified transport adresses. The connection establishment phase provides also for:
  - negotiation of the quality of service;
  - negotiation of the use of expedited data transfer service during the subsequent data transfer phase;
  - transmission of a limited amount of user data.

Figure 14 summarizes the transport service primitives for the connection establishment phase, while figure 15 gives typical operation of connection establishment.

Primitive	Parameters
T-CONNECT request	Called Address, calling address, expedited data option, quality of service, user-data.
T-CONNECT indication	same as T-CONNECT request
T-CONNECT response	Responding address, quality of service, expedited data option, user-data
T-CONNECT confirmation	same as T-CONNECT response

Figure 14: Parameters for connection establishment.

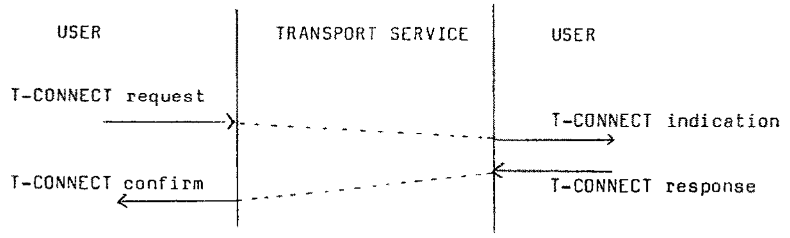


Figure 15: Typical operation of the connection establishment phase.

- The data transfer phase provides for
  - transmission of TSDUs of unlimited lenght in both directions

according to the agreed upon quality of service;

- transmission of expedited TSDUs (up to 16 octets in both directions, if negotiated during the establishment phase.

Flow control is offered independently for both, the expedited and the normal flows. An expedited data may bypass a normal data but a normal data cannot bypass an expedited one.

- The release service allows for terminating the connection at any time. Termination is normally invoked by one of the users (or both simultaneously) but may also be invoked by the service itself in case of errors. Termination is an unconfirmed service which may result in loss of data.

Figure 16 shows typical termination cases of a transport connection termination

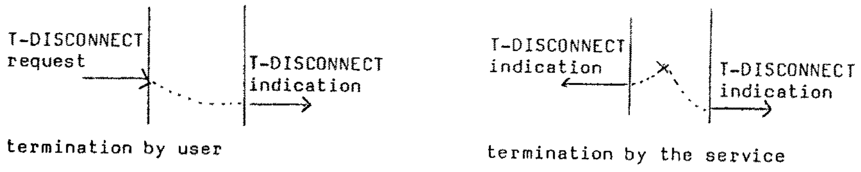


Figure 16: Typical cases of a transport connection termination

9.3.2. The transport protocol

In order to bridge the gap between the network service and the service to be provided to the users, the transport protocol uses the following functions:

- mapping of transport addresses into network addresses,
- assignment of transport connections onto network connections,
- identification of transport connections,
- segmenting TSDUs into TPDUs and reassembling TPDUs into TSDUs,
- implicit flow control (use of the (N-1) flow control) or explicit flow control by means of acknowledgement and credit mechanisms,
- multiplexing of several transport connections onto one simple

network connection,

- explicit disconnection (i.e. disconnecting the transport connection without disconnecting the supporting network connection) or implicit disconnection (via the disconnection of the supporting network connection),
- recovery from errors signalled by the network,
- detection of errors not signalled from the network.

Due to the variety of network services and the differences in the user's requirements, the transport should be able to dynamically adapt the quantity of functions put into operation over a given transport connection. This is done in negotiating the functions to be used at connection establishment time.

In order to simplify the negociation mechanism, the functions have been organized in classes:

- Class 0 is the simplest class including the minimum functionality:
  - connection establishment,
  - data transfer and segmenting,
  - implicit disconnection.
- Class 1 includes class 0 functions and also
  - explicit disconnection,
  - recovery from errors signalled by the network.
- Class 2 includes class 0 functions and in addition those functions necessary for multiplexing:
  - explicit disconnection,
  - connection identification,
  - explicit flow control.
- Class 3 uses class 2 functions and offers additionally recovery from errors signalled by the network layer.
- Class 4 detects and recovers from errors which are not signalled by

the network.

Therefore, class 4 may operate on top of connection-less network (e.g. datagram network) or split a transport connection onto several network connections (use of several network connections for a single transport connection leads to misordering of TPDUs).

The functions included in the classes demonstrate the possible negotiations (see figure 17).

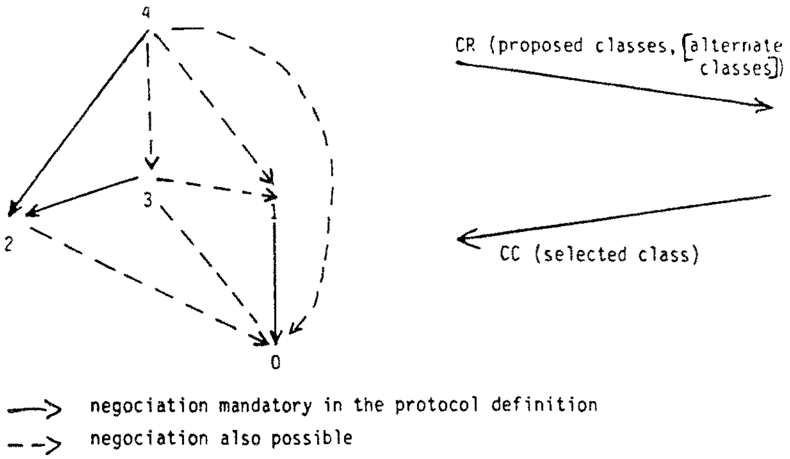

4
3
2
1
0
CR (proposed classes, [alternate classes])
CC (selected class)
negociation mandatory in the protocol definition
negociation also possible

Figure 17: Possible negotiations.

In order to avoid collision on a given connection identifier when establishing two connections simultaneously (and to simplify the implementations), the connection references are established as follows: each party selects one part of the identifier (see figure 18) and communicates it to the partner during the connection establishment phase. Then both partners keep track of the parts which have been locally selected (local reference) and remotely selected (remote reference).

In the data transfer phase only the remote reference is put in the PDUs which are sent (in the 'destination reference field'). When a PDU is received, the destination reference field is used to match with the local reference in order to associate the received PDU with the appropriate connection. In the disconnection phase, both references are exchanged again in order to perform a more secure exchange. This technique gives to implementations all freedom for allocating/releasing references in the most convenient way and avoids 'cross-generation' of systems.

Figure 19 gives the list of the TPDUs and their eventual parameters, while Figure 20 shows some typical protocol exchanges in classes 0,1 and 2.

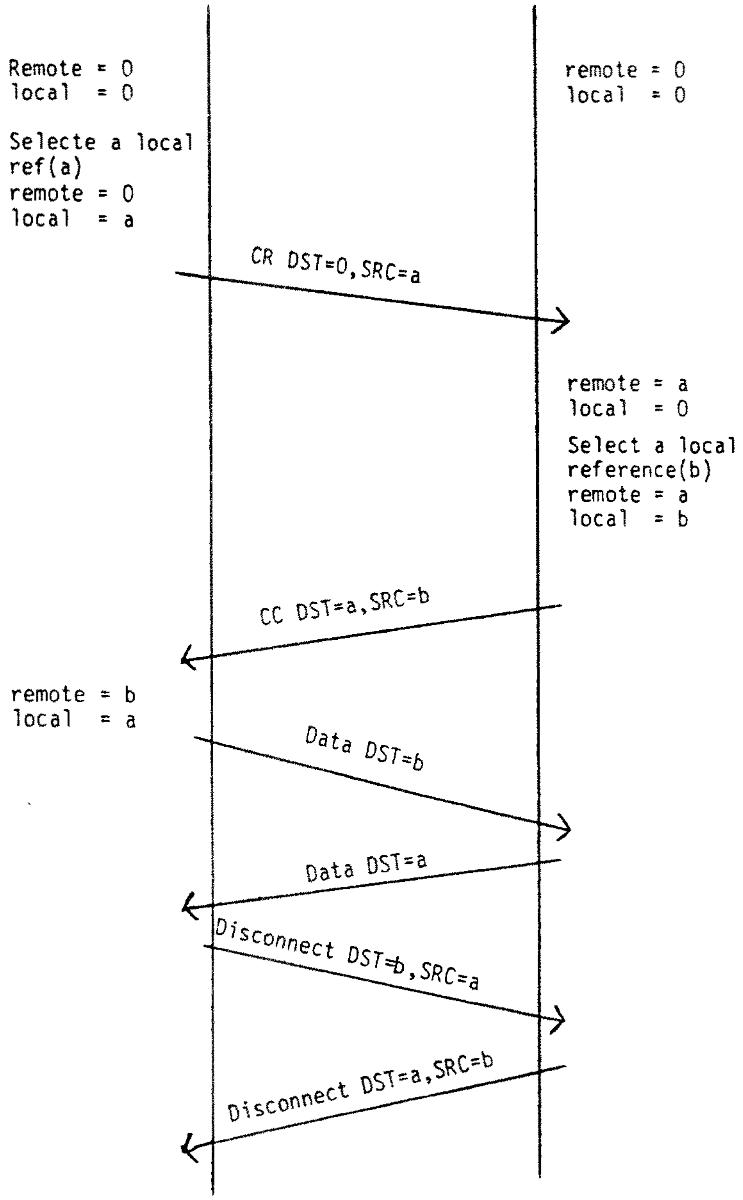


Figure 18: Connection references selection and use.

TPDU's	PARAMETERS
Connection request (CR)	initial credit destination reference, source reference, proposed class and options, alternative class, addresses, quality of service, data
Connection confirm (CC)	initial credit, destination reference, source reference selected class and options, address, quality of service, data
Disconnect request (DR)	destination reference, source reference reason, data
Disconnect confirm (DC)	destination reference, source reference
Acknoledgment (AK)	destination reference, credit, expected TPDU number
Reject (RJ)	=
Data (DT)	destination reference (in class 2,3, and 4) end of TSDU mark, TPDU number in classes other than 0
Expedited data (EX)	destination reference expedited TPDU number, data
Error (ER)	destination reference, reason, text of rejected TPDU.

Figure 19: TPDUS and their parameters.

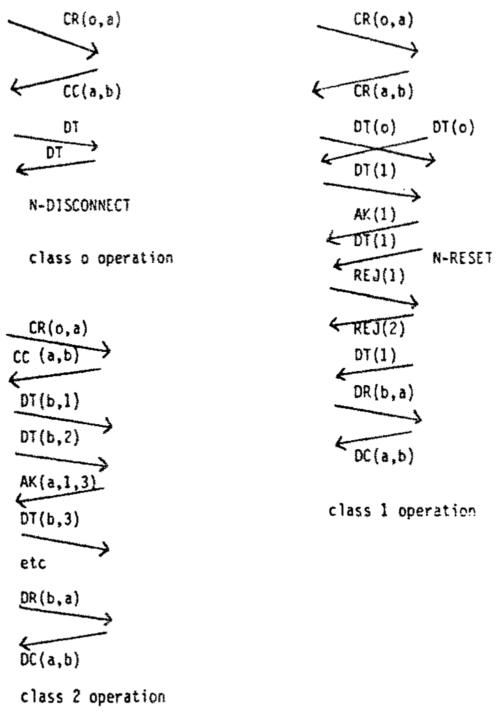


Figure 20 : Typical exchanges on classes 0,1,2.

9.4. Protocol games

This section proposes some "protocol games" based on very simple examples, in order to give to the reader some flavour of the problems which may be encountered when specifying and validating protocols.

For the sake of simplicity, we focus on the following two aspects:

- use of (N−1) flow control service,
- influence of transit delay.

The study is done using a simple protocol invented only for this purpose. When necessary, references are made to existing protocols (e.g. ISO transport and session protocol) where similar problems are found when studying them.

9.4.1. The example

The protocol involves two processes called 'Sender' and 'Receiver'. On request of its user the Sender sends messages of fixed – (short) arbitrary length. The Sender has a variable called X used as follows:

- At initialization time X is equal to zero.
- Before sending a message X is incremented by one.
- If X reaches a bound called M, the Sender detects an error and stops.
- When the Sender receives from the Receiver a RESET message, it resets X to zero.

The Receiver receives messages and passes them to its user.
The Receiver has a variable called Y used as follows:

- At initialization time Y is equal to zero.
- When receiving a message Y is incremented by one
- If Y reaches a bound called N, the Receiver sends a RESET message to the Sender and resets Y to zero.

M and N are parameters of the communication and have values allocated before the communication starts.

The Sender and the Receiver exchange their messages via a medium without error (no loss, no duplication, no misordering).

9.4.2. Notation

In order to facilitate further reference to this protocol we use the following notation:

The Sender and the Receiver are described by means of transitions of the following form:

<event state (predicate) resulting-state message (variable-setting)>

Using the notation, the Sender's transitions are:

S1 : <U-request normal-state (X<M-1) normal state data (X:=X+1)>
S2 : <U-request normal-state (X=M-1) error-state – () >
S3 : <RESET normal-state ( ) normal-state – (X:=0)>

The Receiver's transitions are:

R1 : <data normal-state (Y<N-1) normal-state U-indication
(Y:=Y+1)>
R2 : <data normal-state (Y=N-1) normal-state
RESET&U-indication (Y:=0)>

We represent a global state of the communication as follows : [West 78a, West 78b]

$$\begin{bmatrix} \text{Sender} & \text{Receiver} \\ M(S<-R) & M(R<-S) \end{bmatrix}$$

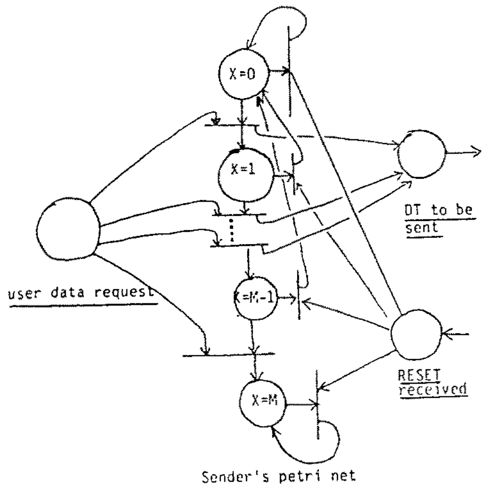
Where Sender and Receiver are the state (and variables) of the processes and M (S<-R) and M (R<-S) are the content of the medium in the specified direction.

In our case the states of sender and receiver are represented by the X and Y variables respectively.

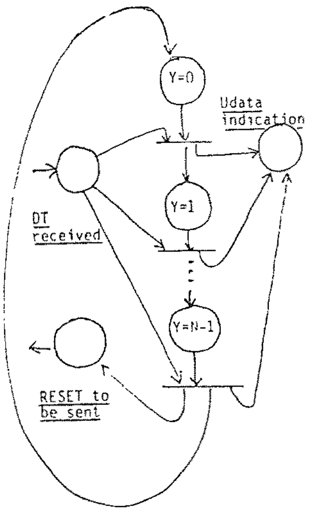
Using this notation the initial state is:

$$\begin{bmatrix} 0 & 0 \\ \text{empty} & \text{empty} \end{bmatrix}$$

We also give a possible Petri net for the Sender and for the Receiver:



Sender's petri net



Receiver's Petri net

9.4.3. Flow control

Although very simple, the protocol is in fact incompletely specified: nothing is said about flow control.

In fact the following implicit assumption is made:

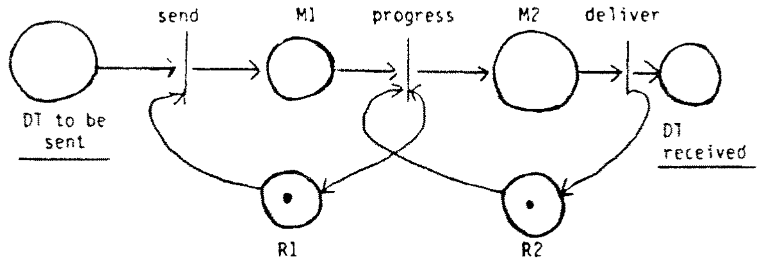
- the receiving user regulates the rate at which the Receiver delivers massage to it,
- the Receiver regulates the rate at which the medium delivers messages to it,
- the medium regulates the rate at which the Sender may pass messages to it,
- and finally the sender regulates the rate at which it accepts messages from its user

Conversely, it is clear that the protocol cannot operate without this assumption; if the Sender can send an infinite number of messages even if the receiver does not process them (i.e. they are in the medium), then the X variable will reach the bound M and the Sender will detect an error.

In general the protocol specification does not include this flow control aspect for the following reason: protocols include control phases based on handshake in which an entity refrains itself from sending an infinite number of messages before waiting for an acknoledgment, and this mechanism hides the above described 'backpressure' flow control.

It should also be noted that a full specification of the communication cannot be obtained without specifying the maximum capacity of the medium.

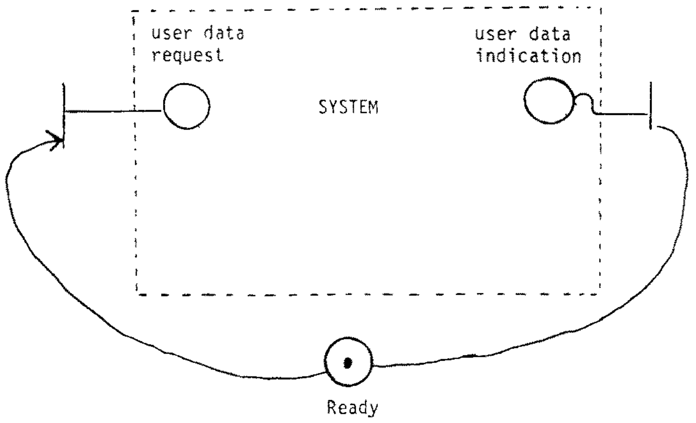
We give here a possible model of medium with a capacity of 2 and the flow control property.



The 'send' transition represents the acceptance of a message by the medium, while 'deliver' models the acceptance of the message by the Receiver.

Places R1 and R2 represent the capacity of the medium, while M1 and M2 model the medium itself (fifo with capacity of 2).

It is now necessary to express the 'back-pressure' flow control between the medium and the Sender (i.e. the Sender shall be refrained to put tokens into the 'DT to be sent' place while the place is busy) and between the Sender and the user, etc... The total capacity of the system is the sum of the capacity of each of its components. Note that, when using Petri nets, a more simple solution is to prevent the sending user to put a new token in the 'userdata request' place until the previous message has not been completely processed by the system.



This model is more simple and guarantees an overall capacity of one (or p if we put p tokens in the ready place). However it is far from the real protocol to be modelled and the equivalence of both models has to be proved.

The ISO transport protocol classes 0 and 1, as well as the ISO session protocol use back-pressure flow control during their data transfer phase.

9.4.4. Transit delay

In general the transit delay is neither specified in protocol, nor used by the validation systems.

The semantic behind this absence of specification of time is sometimes unclear, sometimes an event may appear in the interval (0, infinity).

This is at least the case in Petri nets, but also in the perturbation method as developed at the IBM ZÜRICH Lab [West 78a, West 78b].

Let us consider our example, assuming a flow control and a medium capacity of one. In each direction assume also that M is greater than N but lower than 2 x N.

Let us consider the following global state

$$\begin{bmatrix} X = N & Y = N-1 \\ \_ & \text{MESSAGE} \end{bmatrix}$$

Since the medium is full the only possible next global state is (transition R2).

$$\begin{bmatrix} X = N & Y = 0 \\ \text{RESET} & - \end{bmatrix}$$

Then, the next global state could be either the delivery of the reset message or a production of a new message. The later case will lead to the following global state (transition S1).

$$\begin{bmatrix} X = N+1 & Y = 0 \\ \text{RESET} & \text{MESSAGE} \end{bmatrix}$$

$$\begin{tikzcd}
 & & & \begin{bmatrix} N & 0 \\ \text{RESET} & \end{bmatrix} \arrow[dl] \arrow[dr] & \\
 & & \begin{bmatrix} N+1 & 0 \\ \text{RESET} & \text{mes} \end{bmatrix} \arrow[dl] & & \begin{bmatrix} 0 & 0 \\ - & - \end{bmatrix} \\
 & \begin{bmatrix} N+1 & 1 \\ \text{RESET} & - \end{bmatrix} \arrow[dl] \arrow[dr] & & & \\
\begin{bmatrix} 0 & 1 \\ - & - \end{bmatrix} & & \begin{bmatrix} N+2 & 1 \\ \text{RESET} & \text{mes} \end{bmatrix} \arrow[dl] \arrow[dr] & & \\
 & \begin{bmatrix} 0 & 1 \\ - & \text{mes} \end{bmatrix} & & \begin{bmatrix} N+2 & 2 \\ \text{RESET} & - \end{bmatrix} \arrow[dl] \arrow[dr] & \\
 & & {} & & \text{etc}
\end{tikzcd}$$

Finally the Sender will detect an error and stop. The problem comes from the fact that, since there is no bound of the transit delay for the messages, it is possible to send an 'infinite' number of messages in one direction (Sender to Receiver), while the RESET message sent by the Receiver is not delivered (i.e. is still in transit). Again, this is due to the absence of a handshake in the protocol: in the case of a handshake, the execution of the handshake implies that a message is sent and another received and this represents a bound for the transit delay.

If we assume for the messages a minimum and a maximum transit delay, then the protocol operates properly (with appropriate values for N, M and the bounds of the transit delay).

A similar problem has been discovered in the OSI session protocol in which, in the absence of transit delay, an infinite number of expedited data may bypass a given normal data....

If we go back to our Petri-net, it would be necessary to add a [tmin, tmax] interval to every transition, tmin being the minimum time the transition will wait after being and staying enabled, and tmax being the maximum time the transition may wait after being and staying enabled before firing [Merlin 76].

All transitions will receive an [0,0] interval except

- production of messages by the sending user and acceptance of messages by the receiving user which receive a [0, infinity] interval,
- transitions for progression of messages in the medium which receives a [tmin, tmax] interval.

The resulting Petri-net can be analysed [Berthomieu 83] and will be found correct.

9.4.5. Side effect in flow control

Let us assume now that M> 2*N, and consider that there is no transit delay specified for the messages.

Assume that the medium capacity is one.

Let us consider the following global state.

$$\begin{bmatrix} 2*N & N-1 \\ \text{RESET} & \text{MESSAGE} \end{bmatrix}$$

This global state reached after the Sender has first sent N messages. Then, the Receiver has sent a RESET when receiving the message number N and has set his Y variable to zero. Again the sender has sent N messages before the RESET has been delivered.

Since Y = N−1, the protocol specification says that a RESET message shall be sent back. However, the medium is not ready for accepting a new message. There is a lack of specification in the protocol. Depending on the semantic used for the model of specification, the choice may be:

1. Since the Receiver cannot completely process the received message we consider that the receiver does not withdraw the message from the medium (the sending and receiving flow control is coupled in a loop-back fashion).
2. This is an error.
3. The RESET message is produced by the Receiver and put into a temporary queue in which it waits before being sent.

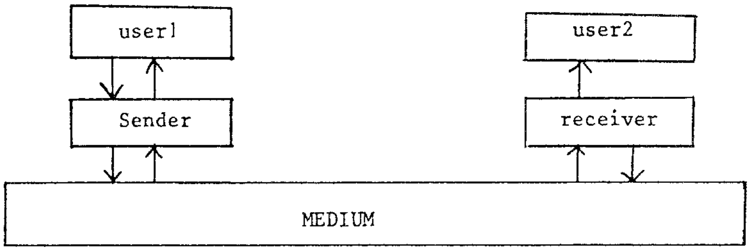
In the case 1 and only in this case the protocol is operating properly.

A similar case has been found in the OSI transport protocol when spurious TPDUs are received and answered (for instance a Disconnect Confirm is sent back for any unexpected received Disconnect Request), regardless of the sending capacity of the network connection.

9.4.6. Protocol correctness versus service conformity

The example below presents a protocol, currently under design by CCITT and called "D-protocol". The protocol is very simple and designed for reliable transmission of messages over an unreliable link which does not signal errors. Only one message is transmitted at a time.

The D-protocol: The model of the D-protocol can be given as follows:



User1 requests of a message in passing to the Sender the DTA request primitive and will eventually receive from the Sender the DTA confirmation which indicates that the Sender is ready for accepting the next request. A parameter of the DTA confirmation also indicates if the message has been successfully transmitted (+ confirmation) or has been (possibily) lost (- confirmation).

At the receiving side there is one service primitive passed by the Receiver to user2: NDTA indication.

The protocol itself makes use of two messages:

DT (Sequence_number, data)	sent by the sender to the receiver
AK (Sequence_number)	sent by the receiver to the sender

The protocol is defined as follows (in pseudo-PDIL)

```
const maxretrans=...;             (* the maximum retransmission *)
      timer_value=...;            (* value to be used for retransmission
                                     timer*)

type sequence_number= 0...1;      (* as for alternating bit *)
      datatype=...;               (* undefined string of octed *)
      conf_type=(positive,negative); (* confirmation *)

interaction  DT = record
                    seq: sequence_number;
                    data: datatype;
             end ;
             AK = record
                    seq: sequence_number;
             end ;
             DTArequest = record
```

```
                data: datatype;
            end ;
            DTAindication = record
                data: datatype;
            end ;
            DTAconfirmation =record
                conf: conf_type;
            end ;

(* sender operation *)

var retrans: integer;                  (* retransmission counter *)
    my_seq: sequence_number;           (* current sequence number *)
    state (ready, waiting);            (* current state *)
    copy_data: data_type;              (* copy of data for retransmission*)
init begin retrans:=0 ;                (* initialize retransmission counter *)
           my_seq:=0 ;                 (* initialize sequence number *)
           to ready;                   (* initial state *)
    end ;
trans when DTArequest                  (* from user *)
  from ready                           (* in ready state *)
     to waiting                        (* next state *)
     send_to_medium_DT(my_seq,
     DTArequest.data);                 (* send *)
     copy_data:= DTArequest.data;      (* next copy *)
     set_timer (timer_value);
  endfrom
endwhen;                               (* in other state, this is a user error *)
when AK                                (* from remote entity via medium *)
  from waiting                         (* in waiting state *)
  provided (AK.seq=my_seq)             (* OK *)
     or (retrans = 0);
     reset_timer; retrans:= 0;         (* reset variable *)
     to ready ;                        (* back to ready*)
     my_seq:=1-my_seq;                  (* ready for next seq. number *)
```

```
        send_to_user_DTAconfirmation (positive);
    endprov
    provided otherwise                (* not OK *)
        reset_timer;
        set_timer (timer_value);      (* a new timer *)
        my_seq:=1-my_seq;             (* change seq *)
        retransm:=retrans+1;          (* retransmit *)
        send_to_medium_DT (my_seq, cpoy_data);
                                      (* again *)
    endprov
    endfrom
endwhen;                              (* in other state this is an error *)

when time_out                         (* timer runs out *)
    from waiting                      (* in waiting state only *)
   provided (retrans=maxretrans)      (* limit reached *)
       retrans:=0;                    (* reset retrans *)
       my_seq:=1-my_seq;              (* ready for next *)
       send_to_user_DTAconfirmation (negative);
                                      (* may be lost *)
    endprov
    provided otherwise                (* limit not reached *)
       retrans:=retrans+1;            (* incremented retrans *)
       set_timer (timer_value);       (* timer needed *)
       send_to_medium_DT(my seq,copy data);
                                      (* retransmission *)
    endprov
    endfrom
endwhen;

(* receiver operation *)

var my_seq:integer;                   (* no state needed *)
init  my_seq:=0;                      (* initialize my sequence *)
trans when DT                         (* from medium *)
```

```
provided (my_seq=DT.Seq)          (* sequence as expected *)
    send_to_medium_AK (my_seq); (* AK *)
    send_to_user_DTAindication (DT.Data);
                                  (* data to user *)
    my_seq:=1-my_seq;             (* ready for next *)
endprov
```

Environment and parameter assumptions: The maxretrans parameter may take any value. The timer–value parameter may take any value provided it is greater than twice the maximum transit delay of the medium. The medium may transmit or loose any message provided the message is either delivered before the maximum transmit delay or never. Provided the above described conditions are respected, there is only one message in transit at a time, therefore, the question whether the medium may deliver the message out of order is irrelevant.

Protocol validation: The validation of the protocol has been done (using LISE) and shows the following:

- There are no deadlocks or unspecified receptions.
- The global state automata is connex and contains no unproductive productive cycles.

Service conformity: Although the protocol validation has been successfully accomplished, it remains to be proved that the protocol provides the service it has been designed for.

The service can be defined as follows:

- For every NDTArequest issued by the user, the data is either delivered at the other end (NDTAindication) or lost.
- A NDTAconfirmation (positive) is never delivered if the data is lost.
- A NDTAconfirmation (negative) may be delivered even if the data has not been lost, but on this case, is passed to the user after the data has been delivered to the remote user.
- If the data is lost a NDTAconfirmation (negative) is always passed to

the user.

- If the data is not lost a NDTAconfirmation (positive) is always passed to the user.

In other words, the only valid sequence of service primitive is defined as:

```
valid-sequence::= NDTArequest.<result>.<valid-sequence>
result::= NDTAindication. NDTAconfirmation (positive)
          |NDTAindication. NDTAconfirmation (negative)
          |NDTAconfirmation (negative)
```

The service conformity checking can be done by using "global variables" and one "invariant transition".

A variable is said to be global if it can be accessed by the sender and receiver side. An invariant transition is a spontaneous transition which shall never be enabled, otherwise an error is detected (in other words the predicate representing the guard of the transition is of the form "provided invariant-is-false").

The service conformity can be checked as follows: the two following global variables are introduced.

```
diff : integer        (* the difference between the number of data sent
                       and the number of data passed to the user *)
receiv : boolean      (* true if a message has been delivered *)
```

- The transitions are modified as follows:

```
  -  sending DT:             diff:=diff+1;
                              receiv:=false;
  -  passing data to user:   diff:=diff-1;
                              receiv:=true;
  -  passing NDTAconfirmation (negative) to the user
                              if (not receiv) then diff:=diff-1;
```

- The following invariant transition is introduced

```
                              provided (diff <0 or diff>1) to error-state;
```

Running the validation with LISE has shown that a service violation is possible (i.e. a message lost and positive confirmation returned).

9.4.7. Conclusion

A number of other "games" can be constructed with these protocols or with variants, but this is not our talk... Our goal was only to illustrate by a few examples the need for:

- A precise specification of a protocol including the relationship between the (N)-protocol and the (N+1)-user, and the use of the (N-1)-service.
- A precise and well known semantic when the validation of the protocol is done. Otherwise problems may be detected only at implementation time.
- A precise definition of the service the protocol should provide.

There exists a large variety of techniques for specifying and analyzing protocols. The most important, from the user's point of view, seems to be how to deal with the four steps of the design procedure: specification - validation - implementation - maintenance. The tools which have been presented along these lectures are probably not the most advanced and efficient for a specific phase, however they present the advantage to be fully operational, to have been experimented with a large number of 'real' protocols and to provide aid for each of the four above mentioned phases. Conversely, we must admit that the techniques presented here have the drawback to work at a "low" level of abstraction and to use validation by enumeration rather than verification by proof....