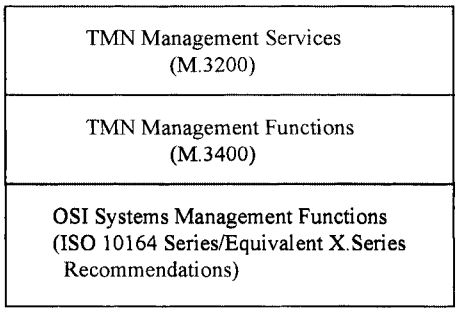


Figure 3-3
Basis for TMN systems management services.

- *Customer administration:* Concerned with fulfilling customer requirements and interfacing with customers on the telephone services offered. Customer administration needs to transmit these telephone service—related data, such as the services and features, to the network administration to satisfy customer requirements. This MS includes service provisioning management, configuration administration, fault administration, charging administration, complaints administration, quality of service administration, and traffic measurement administration. Some of the customer administration functions are service activation, customer request to make changes to the services, customer request for information about the status of services and network resources, and reporting of customer trouble to the trouble ticket mechanism.
- *Network provisioning management:* Encompasses the functions of management of processes for providing new and traditional services to customers in a very efficient and proactive manner. To provide new resources or features, an expansion of the existing resource capacity may be required.
- *Workforce management:* Responsible for the deployment of appropriate field staff to perform maintenance, repair, and installation. This MS may also involve deployment of field staff in customer premises.
- *Tariff, charging, and accounting administration:* Involves the whole gamut of billing and accounting management of customers. Some of the functions include billing the customers at regular intervals; taking action on unpaid bills; and resolving customer complaints on billing, including errors, changes in tariffs based on customer choices, detection of fraud using traffic analysis, and so on. For new customers, billing and pricing processes have to be set up. Tariff, charging, and account administration also require usage measurement and testing.

Billing starts with service activation and necessary paperwork that has to be generated with the service activation.

- *Quality of service and network performance administration:* Customers need consistent telecommunications services with good transmission quality and assured performance. The QOS and network performance administration includes root cause elimination by investigation, interviewing, analysis, and testing. Some functions of this MS are traffic quality assurance; performance quality assurance; and reliability, availability, and survivability (RAS) quality assurance. Information from NEs on fault localization, network fault localization, and service outage reports provide input for RAS quality assurance.
- *Traffic measurement and analysis administration:* Sometimes, traffic load on a network may be larger or smaller than planned. Also, adding new services can change the traffic pattern. In such scenarios, a traffic administration process is required to resolve the congestion or underutilization problems. Traffic administration also includes a traffic measurement process.
- *Traffic management:* Kicked off to solve current traffic problems as a result of failures and outages of telecommunications resources, abnormal increases in traffic load, and so on. For traffic management, performance monitoring and performance management functions are required.
- *Routing and digit analysis administration:* Required for verifying routing information in an exchange, changing routing information in tables, and switching routing tables on a time schedule.
- *Maintenance management:* When a failure or a problem is detected in any resource in a network, a trouble ticket is opened. The failure/problem rectification is tracked with the trouble ticket opened as a reference. Here, mean time between failures is also helpful. Maintenance management includes proactive maintenance, network detected trouble, fault localization, and fault correction.
- *Security administration:* Concerned with the security of switches, customers, and billing and accounting information. It also deals with establishing and changing customer privileges, detection of security violations, and taking steps to prevent security violations.
- *Logistics management:* Required for offering telecommunications services to customers and for achieving improvements in the availability of equipment in a telecommunications network. Materials management is responsible for procuring items and making them available to the transmission equipment and other equipment in the telecommu-

nications network in a timely manner. Logistics management enables timely deployment of resources to meet customer and internal demands, keeping the cost factor in focus.

3.2.1 Mapping of SMFAs and TMN Management Function Set Groups

As we have seen in Chapter 1, SMFA is an OSI systems management term. The same term is also popular in TMN. Figure 3-4 shows the mapping of SMFAs to different TMN Management Function Set Groups. We discuss each of the TMN management function set groups in detail in the following sections.

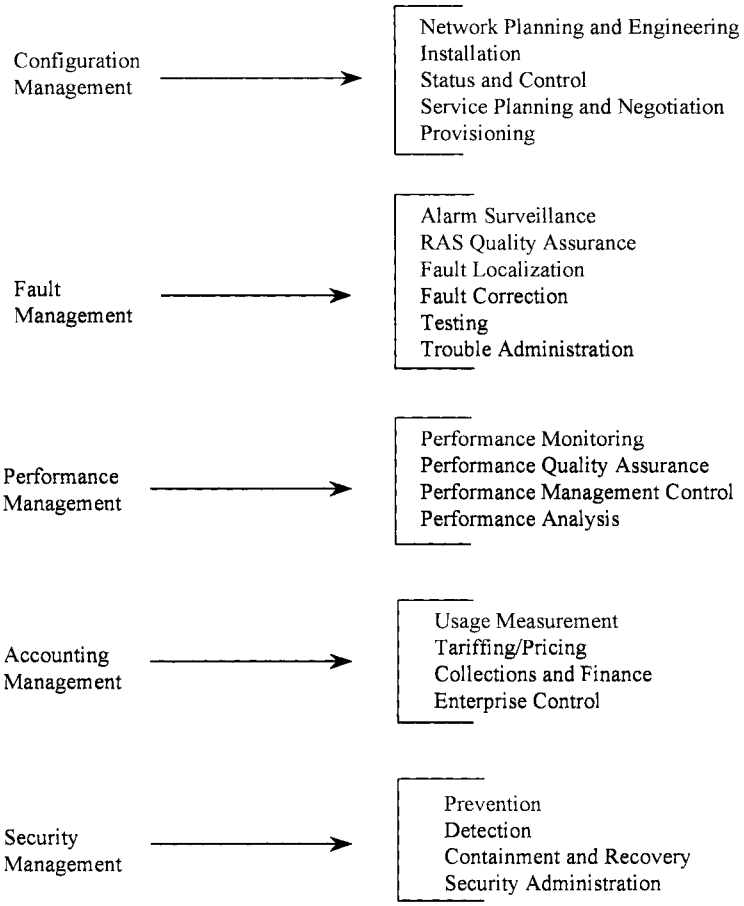


Figure 3-4 Mapping of SMFAs and TMN management function set groups.

3.3 TMN Management Function Sets and Management Functions

In this section we examine how TMN management functions are relevant to network management. TMN management functions are defined in ITU-T Recommendation M.3400, TMN Management Functions. The material included here is patterned on the lines of M.3400. For details on each of the individual function sets, refer to M.3400, which has detailed descriptions.

3.4 Performance Management

One of the primary performance management functions is to collect performance-related data on the telecommunications networks and equipment. Once these data are collected from the resources in a manager or OS from network elements, the data have to be analyzed. The performance data—including the reports on the data, especially the bottlenecks—have to be identified and the problems have to be corrected. Performance management also includes traffic management. ITU-T Recommendation Q823 (Reference 3.6) defines many managed object classes. These should be utilized in the design of the traffic management application.

Performance management also involves collecting QOS-related data and improving the QOS. In addition, performance management includes the performance-related issues of NEs. Note that in the explanations given here, *NEs* refers to one or more NEs.

QOS is measured by characteristics such as the rate of information transfer; the probability of system failure, storage failure, and communication disruption; and the latency. QOS is measurable at the service access point and is quantified in terms of user-perceivable effects.

QOS is an important factor in the TMN. QOS is applicable to the following parameters:

- Customer connection establishment
- Customer connection retention
- Customer connection quality
- Billing integrity
- Keeping and examining logs of system state histories

- Cooperation with fault management to establish source of failure of a resource
- Cooperation with configuration management to change routing and load control parameters
- Performing tests to monitor QOS parameters

However, many issues are involved in the collection of performance data. As an example, if the performance data is collected in short intervals, then the network traffic will increase, impacting network performance. On the other hand, if too few data are collected, then usability of those data will be severely affected. Therefore care must be taken to use the right kind of schedule to collect the performance-related data.

There are four function set groups in performance management. These are discussed in the following text.

3.4.1 Performance Quality Assurance

The function of performance quality assurance is to establish processes to meet the customer's needs with respect to the performance management. Performance quality assurance has the following function sets:

- QOS performance goal setting
- QOS performance assessment
- Network performance goal setting
- Network performance assessment
- Subscriber service quality criteria
- NE performance assessment
- Data integrity check

3.4.2 Performance Monitoring

The function of performance monitoring (PM) is to continuously collect performance-related data on system, network, or service activities. PM is slightly different from alarm surveillance. Alarm surveillance is related to acute fault conditions, whereas performance monitoring is related to the very low-rate or intermittent error conditions not detected by alarm surveillance. PM includes the following function sets:

- Performance monitoring policy
- Network performance monitoring event correlation and filtering
- Data aggregation and trending
- Circuit-specific data collection
- Traffic status
- Traffic performance monitoring
- NE threshold crossing alert processing
- NE trend analysis
- Performance monitoring data accumulation
- Detection, counting, storage, and reporting

3.4.3 Performance Management Control

Performance management control is responsible for controlling the performance of a telecommunications network. Network traffic management includes traffic control, which affects routing of traffic and processing of calls. Transport performance includes functions such as setting thresholds and collecting performance data. Performance management control includes the following function sets:

- Network traffic management policy
- Traffic control
- Traffic administration
- Performance administration
- Execution of traffic control
- Audit reporting

3.4.4 Performance Analysis

Performance analysis is involved in the analysis of performance data of an entity. It supports the following function sets:

- Recommendations for performance improvements
- Exception threshold policy
- Traffic forecasting
- Customer service performance summary

- Customer traffic performance summary
- Traffic exception analysis
- Traffic capacity analysis
- Network performance characterization
- NE performance characterization
- NE traffic exception analysis
- NE traffic capacity analysis

3.5 Fault Management

Fault management is responsible for detecting and isolating abnormal conditions that affect the operation of a telecommunications network and its environment. Fault management also includes quality assurance measurement for RAS. Effective fault management may require errors to be logged in a database. The fault management functional area covers the fault management function set groups presented in the next sections.

3.5.1 RAS Quality Assurance

RAS quality assurance establishes guidelines for reliability of other fault management—related functions and the design of redundant equipment. RAS quality assurance includes the following function sets:

- Network RAS goal setting
- Service availability goal setting
- RAS assessment
- Service outage reporting
- Network outage reporting
- NE outage reporting

3.5.2 Alarm Surveillance

Alarm surveillance provides the capability for real-time monitoring and interrogation of NE failures. When there is a failure in an NE, it reports the failure, sometimes along with the nature and severity of the fault. If

there is intelligence in an NE, it will send notifications to the manager or OS with the details of failure. In simple cases, the NE may just indicate that a failure has occurred. These details of the failure may be reported at the time of its occurrence or logged for future use. An alarm, because of a failure, may lead to some other management actions within an NE.

At this juncture it is appropriate to look into the distinctions between an alarm and an event. An *alarm* is a notification for a specific event indicating a problem condition. An alarm may or may not represent an error. An *event* is an instantaneous occurrence that changes at least one of the attributes representing the global status of an object. This status change may be persistent or temporary, and it may be monitored by alarm surveillance or performance measurement functionality. Events may or may not generate reports, and may be spontaneous or planned.

For alarm surveillance, an NE must permit monitoring of the alarm in real time or in a scheduled manner, and the NE must be enabled to query on the alarm conditions and allow logging and retrieval of the historical alarm information. There is a separate ITU-T recommendation for alarm surveillance (Reference 3.5) that should also be used while designing alarm surveillance. Alarm surveillance includes the following function sets:

- Alarm policy
- Network fault event analysis, including correlation and filtering
- Alarm status modification
- Alarm reporting
- Alarm summary
- Alarm event criteria
- Alarm indication management
- Log control
- Alarm correlation and filtering
- Failure event detection and reporting

3.5.3 Fault Localization

When a fault occurs, it is necessary to identify its location. This may require tests to pinpoint the problem and its cause. This functionality is the domain of the fault localization function. Fault localization has the following function sets:

- Fault localization policy
- Verification of parameters and connectivity
- Network fault localization
- NEs fault localization
- Running of diagnostics

3.5.4 Fault Correction

Fault correction is responsible for repairing or replacing the faulty equipment or using redundant equipment to replace the faulty equipment after a fault has been detected. This area has plenty of scope for automation, thereby improving productivity for telecommunications service providers and bettering the customer satisfaction. Fault correction has the following function sets:

- Management of repair processes
- Arrangements for repairs with customer
- Scheduling and dispatching administration of repair forces
- NE fault correction
- Automatic restoration

3.5.5 Testing

Testing of equipment to check its characteristics or analysis of circuits is done routinely. The management application is responsible for receiving requests from an NE to conduct tests and reporting the results of the tests. The tests and the processing of test results are done in the NE. These test results may be sent to management applications immediately or on a delayed basis. In another method of testing, the management application requests access for testing an NE. The main testing and processing is done by the management application. Testing includes the following function sets:

- Test point policy
- Service test
- Circuit selection, test correlation, and fault location
- Selection of test suite
- Test access network control and recovery

- Test access configuration
- Test circuit configuration
- NE test control function
- Results and status reporting
- Test access path management
- Test access

3.5.6 Trouble Administration

Trouble administration is related to the administrative aspects of trouble reports generated by trouble tickets. Trouble reports are generated due to problems reported by customers or due to proactive equipment failure detection checks. Trouble administration has the following function sets:

- Trouble report policy
- Trouble reporting
- Trouble report status change notification
- Trouble information query
- Trouble ticket creation notification
- Trouble ticket administration

3.6 Configuration Management

The scope of configuration management covers areas such as collecting data from NEs, providing data to NEs, and exercising control over data collection from NEs.

3.6.1 Network Planning and Engineering

The network planning and engineering function is devoted to determining the needs for growth in capacity and the introduction of new technologies. It also involves evaluating alternate plans, and the output of this function becomes input to provisioning to implement the plan for introducing new technology and services. Network planning and engineering includes the following function sets:

- Product line budget
- Supplier and technology policy
- Area boundary definition
- Infrastructure planning
- Management of planning and engineering process
- Demand forecasting
- Network infrastructure design
- Access infrastructure design
- Facility infrastructure design
- Routing design
- NE design

3.6.2 Installation

The installation function set group is responsible for installing the telecommunications network, equipment, and NEs. Installation may also involve extending or shrinking the telecommunications network; installing hardware and software and initial data loading; and testing that a piece of equipment works in a desired fashion and meets the requirements. Installation includes the following function sets:

- Procurement
- Management of installation
- Contracting
- Real estate management
- Arrangement of installation with customer
- Network installation administration
- Materials management
- Scheduling and dispatch administration of installation
- Installation completion report
- Software administration
- NE installation administration
- Loading software into NEs

3.6.3 Service Planning and Negotiation

Service planning and negotiation is concerned with new services, upgrading of services and features, and disconnecting services. Service planning and negotiation has the following function sets:

- Service planning
- Service feature definition
- Marketing
- Management of sales process
- External relations
- Customer identification
- Customer need identification
- Customer service planning
- Customer service features
- Solution proposal

3.6.4 Provisioning

Provisioning involves the procedures required to bring equipment into service. It does not include installation. Provisioning may also control the state of a unit—such as in service, out of service, on standby, reserved—and other important parameters. We also briefly covered the topic of provisioning in Chapter 2, Section 2.4. Provisioning includes the following function sets:

- Provisioning policy
- Materials management policy
- Access route determination
- Directory address determination
- Leased circuit route determination
- Request for service
- Service status administration
- Network resource selection and assignment
- Interexchange circuit design
- Access circuit design

- Leased circuit design
- Facility design
- Management of pending network changes
- Network connection management
- Circuit inventory notification
- Circuit inventory query
- NE configuration
- NE administration
- NE database management
- Assignable inventory management
- NE resource selection and assignment
- NE path design
- Loading program for service features
- NE inventory notification
- NE inventory query
- Management of pending changes in NEs
- Storage of parameters and cross-connects
- Storage and execution of service
- Self-inventory

3.6.5 Status and Control

Status and control has the responsibility of monitoring and controlling the activities of NEs, checking and changing the service status of an NE, and initiating diagnostic tests within the NE. Status and control can also be part of routine maintenance. Status and control has the following function sets:

- Priority service policy
- Priority service restoration
- Message handling systems network status
- Leased circuit network status
- Transport network status
- NE status and control
- Access to state information
- Notification of state changes by NEs

3.7 Accounting Management

Accounting management is devoted to the measurement of costs to the network service provider and charges to customers for the usage of network services and features. The description of the function set groups in accounting management is furnished in the following sections.

3.7.1 Usage Measurement

The data for charging customers is collected from NEs and stored in an OS. Data collection for charging and processing of data has to be reliable and, sometimes, has to be done in real time. There is also a need to keep records of billing to resolve discrepancies that may arise later. Usage measurement includes the following function sets:

- Planning of the usage measurement process
- Management of the usage measurement process
- Usage aggregation
- Service usage correlation
- Service usage validation
- Usage distribution
- Usage surveillance
- Usage error correction
- Usage testing
- Measurement rule identification
- Network usage correlation
- Short-term usage storage
- Long-term usage storage
- Usage accumulation
- Usage validation
- Administration of usage data collection
- Usage generation

3.7.2 Tariffing/Pricing

A *tariff* is a set of data used to determine the charges for services used. A tariff may depend on the service, origination and destination, tariff period, and day of call. A tariff has the following function sets:

- Pricing strategy
- Tariff and price administration
- Costing
- Settlements policy
- Feature pricing
- Provision of access to tariff/price information
- Rating usage
- Totaling usage charges

3.7.3 Collections and Finance

The collections and finance TMN functional set group includes administration of customer accounts, informing customers on payment dates, payment amount, and collection of payments. This function set group includes the following function sets:

- Planning of the billing process
- Management of the billing process
- General accounting operations
- General ledger
- Accounts receivable
- Accounts payable
- Payroll
- Benefits administration
- Pension administration
- Taxation
- Human resources
- Invoice assembly
- Sending invoices
- Customer tax administration
- In-call service request
- Storage of invoice
- Receipt of payment
- Inquiry response
- Collections

- Customer account administration
- Customer profile administration

3.7.4 Enterprise Control

Enterprise control is responsible for the proper financial management of an enterprise. It also includes identifying and ensuring financial accountability of officers and the flow of funds between the enterprise and its owners and creditors. Enterprise control includes checks and balances needed for a smooth financial operation of an enterprise. Enterprise control has the following function sets:

- Budgeting
- Auditing
- Cash management
- Raising equity
- Cost reduction
- Profitability analysis
- Financial reporting
- Insurance analysis
- Investments
- Asset management
- Tracking of liabilities

3.8 Security Management

Security management is concerned with security in communication between systems, between customers and systems, and between internal users and systems. Security services include authentication, access control, data confidentiality, data integrity, and nonrepudiation. These security services are defined in ITU-T Recommendation X.800 (Reference 3.8). Security services are also required for event detection, security audit trail management, and security recovery. Security violations such as access by unauthorized users and tampering with important data are reported to appropriate security violation tracking layers.

3.8.1 Prevention

The prevention function set group is related to activities such as prevention of intrusion by unauthorized users. Prevention also includes access control and has the following function sets:

- Legal review
- Physical access security
- Guarding
- Personnel risk analysis
- Security screening

3.8.2 Detection

Detection is both a proactive and an after-the-fact function. It is responsible for guarding against possible intrusion by keeping track of unusual activities, and has to take into account both detecting and tracking intrusions. Detection has the following function sets:

- Investigation of changes in revenue patterns
- Support element protection
- Customer security alarm
- Customer profiling
- Customer usage pattern analysis
- Investigation of theft of service
- Internal traffic and activity pattern analysis
- Network security alarm
- Software intrusion audit
- Support element security alarm reporting

3.8.3 Containment and Recovery

Containment and recovery has to do with prevention of intrusion, repair of damage, and recovery after intrusion and security violations have occurred. Containment and recovery has the following function sets:

- Protected storage of business data
- Exception report action
- Theft service action
- Legal action
- Apprehending
- Service intrusion recovery
- Administration of customer revocation
- Protected storage of customer data
- Severing external connections
- Network intrusion recovery
- Administration of network revocation list
- Protected storage of network configuration data
- Severing internal connections
- NE intrusion recovery
- Administration of NE revocation list
- Protected storage of NE configuration data

3.8.4 Security Administration

Security administration has to do with managing, planning, and administrating security-related policies and security related data. Security administration has the following function sets:

- Security policy
- Disaster recovery planning
- Management guards
- Audit trail analysis
- Security alarm analysis
- Assessment of corporate data integrity
- Administration of external authentication
- Administration of external access control
- Administration of external certification
- Administration of external encryption and keys
- Administration of external security protocols

- Customer audit trail
- Customer security alarm management
- Testing of audit trail mechanism
- Administration of internal authentication
- Administration of internal access control
- Administration of internal certification
- Administration of internal encryption
- Network audit trail management
- Network security alarm management
- NE audit trail management
- NE security alarm management
- Administration of key for NEs
- Administration of key by an NE

3.9 Implementation Notes

Development of network management solutions can be broadly divided into the following categories:

- Providing network management basic infrastructure such as implementing manager and agents.
- Providing network management application solutions. Note that in M.3020 (Reference 3.3), management application is also known as *management services*. As an example, we want to add new network management functionality such as trouble ticket application and integrate with help desk features. In this case, there are two scenarios. In one scenario, we will buy the software packages available for trouble ticket and help desk features. After buying these packages, we integrate the software packages and tailor them to meet the specific requirements of an organization. In the other case, we develop all the software packages in house and do the integration as well.
- Providing enhancements to the existing network management application functions. In this case, we are already using a network management application, but we are extending the functionality of the application by adding new features. As an example, we have the trouble ticket and help desk features. Now, we want to add some automa-

tion features to the trouble ticket and help desk features, such as automatic paging of the appropriate repair technician and sending an e-mail with all the details of a given problem so the technician can attend to and rectify the problem.

In all cases, the first activity involved in development is to gather the user needs. From the user needs, detailed requirements are prepared. The more detailed and stable the requirements are, the fewer problems will be encountered in the design and development stages. In most of the TMN implementations, TMN solutions are staged into different phases/releases for convenience and easy development. Therefore the requirements should be broken down into different stages.

From the list of requirements follow design, development, and testing. Testing includes system test, conformance tests, and acceptance tests. This is the basic flow of activities for developing network management solutions. Note that this development scenario is almost similar to normal development activities encountered in any software development. However, in telecommunications, due to regulatory environments, the quality requirements are stringent.

Now we examine the specific development cycle and activities involved in providing network management solutions. First, identify and prepare the list of telecommunications resources to be managed. Also, identify the management information needed and the operations to be performed on the telecommunications resources. From this, prepare a list of requirements on how these resources are to be managed.

From the list of requirements, prepare the list of TMN management services and management functions. We have already examined what the management services and management functions mean. After this step, develop a management information model—this process is also known as *object modeling*. The management information model is a means to specify interfaces between resources and managing systems. This is known as the *top-down approach*. This approach is time consuming, but it permits refinement of object classes in an iterative manner.

In another approach, known as the *bottom-up approach*, managed object classes are modeled from the resources. This approach saves time, but it has the limitation of creating complex managed object classes and not permitting easy refinement of managed object classes. GSM 12.00 (Reference 3.8) suggests a pragmatic approach, which is a combination of the top-down and bottom-up approaches. We have deliberately introduced the pragmatic approach here, as it is an interesting alternative to the top-down and bottom-up approaches. For more details on the pragmatic approach, refer to Reference 3.8.

One of the principal activities of information modeling is the development of managed object classes with associated attributes, actions, behaviors, and notifications. The information model also contains relationships between managed object classes. Entity relationship (E-R) diagrams represent the relationship between managed object classes. The managed object classes represent the management aspects of the telecommunications resources. The telecommunications resources to be managed usually reside in managed systems.

The information model defines the interface between resources and the managing system and also the messages exchanged between resources and managed systems. CMIP or SNMP protocols can be used to exchange management information. We have specifically used the term *exchange* to convey that the management information flow can be from a managed system to a resource and also from a resource to a managed system.

From the management information model, it is clear when and how a resource has to report or reply to the managing system. And from the managing system's point of view, it is clear how the managing system will receive notifications and how it will process management operations such as CMIS M-SET, M-GET, and others. How the managing system analyzes the received information and reacts to the management operations is not included in the standardization process.

While designing object classes, there are many options. One of the options is to use the standard managed object classes defined in ANSI standards, GSM standards, ITU-T recommendations such as M.3100, and X.721 standards. In addition, there are many standard documents related to specific activities. These standard documents also define managed object classes. As an example, X.740 defines the managed object class securityAuditTrialRecord. Thus this managed object class can be used without changes if there is a need to record security-related activities.

Sometimes, these managed object classes do not satisfy the specific requirements. In such cases, one has to define one's own object classes. If one goes for proprietary definition of managed object classes, one has to define the inheritance hierarchy and the containment hierarchy as well. The definition of managed object classes is done using ASN.1 and GDMO definitions. In the case of TMN solutions using Internet network management, SMI for Internet has to be used for defining managed object classes.

The management information schema is derived from information model. It represents the view of the information model presented by a managed system to a managing system. The management information schema contains all managed object classes that are visible to a managing system, the naming hierarchy between managed object classes in the man-

aged system, and all possible cases of information exchanged between a managed system and managing system.

A strategy to instantiate object classes is required. Note that in the following discussions on object instances, the way managed object classes are defined largely influences how object instances can be created. Let us take some examples of how to instantiate object instances. For bit error rate (BER) testing to be performed on a resource, one instance of the BER test object class is required in each NE and the operations system (OS). That means an OS will have as many instances of the BER object classes as there are NEs. As an example, if three NEs are to be subjected to BER testing, the OS will have three instances of the BER test object class (Figure 3-5). With this arrangement, all NEs can be subjected simultaneously to BER testing.

The design of the BER test object class may be changed such that we have only one object instance of the BER test object class in the OS and each NE has one object instance of each BER test object class. In this case it is possible to perform BER testing on only one NE at a time.

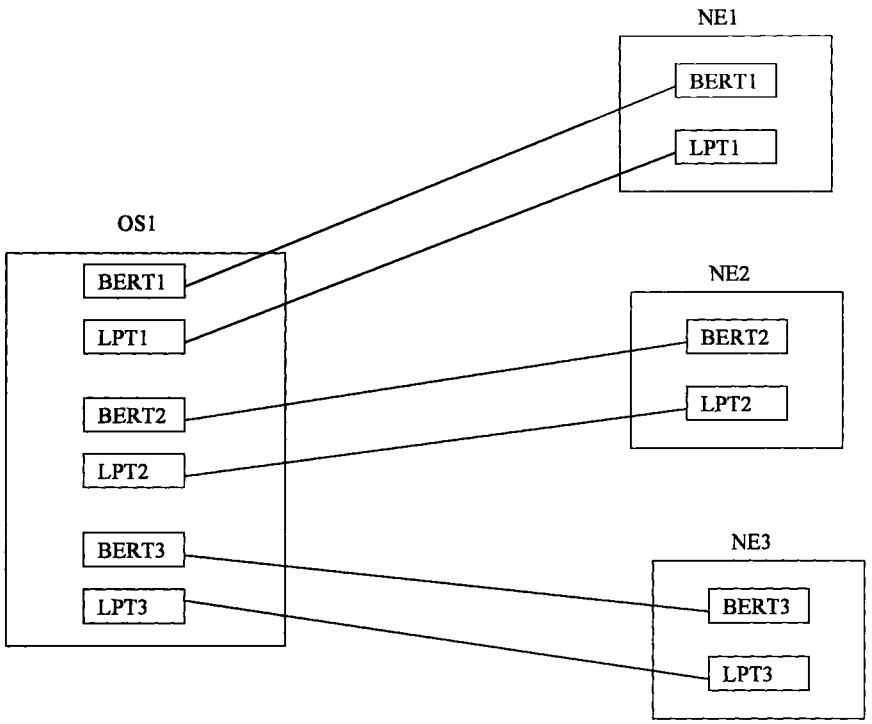


Figure 3-5
Instantiating object classes—separate BER and loopback tests.

However, if another test such as a loopback test is also to be performed on the resource, then we need another set of loopback test object instances in the NE and the OS (Figure 3-5).

We have to be careful while creating object instances. If too many are created, it may cause a performance bottleneck. To reduce the number of object instances, we can create a single managed object class for BER test and loopback test (Figure 3-6). In this case we need only one set of object instances each in an OS and an NE. However, this design imposes a restriction that a BER test and a loopback test cannot be performed at the same time. Besides, if a managed object class is complex and performs a large number of activities, design and implementation of the managed object class can present difficulties. In such cases, a trade-off is required.

Along with the design of the information schema, we have to determine the communication requirements, such as the type of transactions, file transfer, file access, and so on. Also, there can be requirements of throughput, reliability, restrictions on naming, and transit delay. The type of communication stack required should also be determined at this stage.

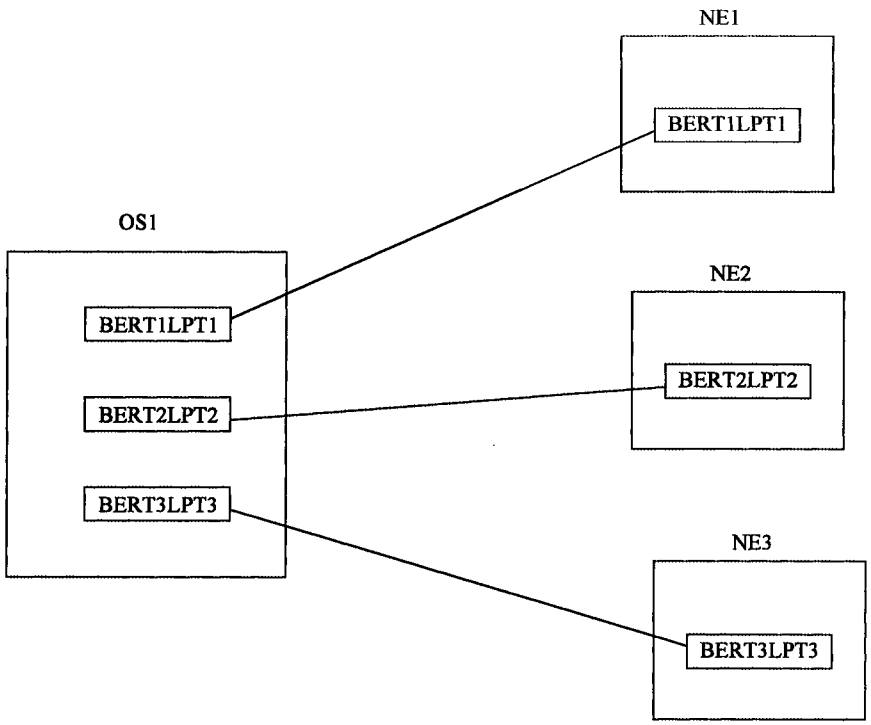


Figure 3-6
Instantiating object classes—combined BER and loopback tests.

As an example, sometimes it may not be necessary to have a full OSI seven-layer stack. Instead of the OSI seven-layer stack, a partial OSI stack with fewer than seven layers may be enough to meet the needs of the management applications.

Let us take the case of developing a performance management application. As an example, for performance management, it is necessary to take into consideration ITU-T Recommendation M.3400, Workload Monitoring Function, X.739 (ISO 10164-11), Summarization Function, X.738 (ISO 10164-13), and Q.822, Description for the Q3 Interface—Performance Management (Reference 3.4). The managed object classes required for the performance management must be a collection of managed object classes defined in the standards.

The workload monitoring function defines object classes for monitoring performance, such as scanner, monitorMetric, gaugeMonitor, meanMonitor, movingAverageMeanMonitor, ewmaMeanVarianceMonitor, and ewmaMeanPercentileMonitor.

Similarly, the managed object classes defined in the summarization function aid in the preparation of summary reports on one or more attributes of observed managed objects. The managed object classes defined in the summarization function document are: simpleScanner, dynamicScanner, dynamicSimpleScanner, ensembleStatisticScanner, heterogeneousScanner, bufferedScanner, homogeneousScanner, meanScanner, meanVarianceScanner, minMaxScanner, and percentileScanner. The managed object classes for the workload monitoring function and the summarization function are explained in the respective standards or in Reference 3.9. Therefore they are not explained further.

ITU-T recommendation Q822 also defines managed object classes for performance management, such as currentData, historyData, and thresholdData. currentData is a subclass of scanner. An instance of currentData contains the current performance data. An instance of currentData is contained in the managed object being monitored and has performance data of the containing managed object. The managed object class historyData is a subclass of *top*. The attributes of the historyData managed object are a copy of the attributes in the currentData object at the end of a recording interval. As a result, current data becomes history data at the end of a recording interval. Again, to collect the current data, a new instance of historyData is created. The thresholdData managed object class supports the attributes for threshold settings for performance monitoring. The thresholdData is a subclass of *top*.

While developing the performance management application, it is necessary to identify the resources on which performance data must be collected. Then it is necessary to decide how and in what frequency per-

formance data have to be collected. Here it is necessary to check whether object classes that have been defined in the standards can be reused. If these standards do not satisfy the requirements, then one has to define one's own object classes. Reference 3.9 contains more details on how to develop different SMFAs.

NEs may require one or more OSs. If there are too many NEs to be managed by a single OS, then more than one OS will be required. In this case, we will become involved with distributed network management. We will look into distributed network management issues in Chapter 11.

In another case, the type of OS required is governed by the management functions to be provided. As we have seen in Chapter 2, an OS may be used to provide element management, network management, service management, or business management layer functions. As an example, for service management functions such as billing and customer support, a service management OS will be required.

Some of the tricky issues in the implementation are how to transfer data from NEs to an OS. In some cases, such as that of performance data, a large amount of data is to be handled. Scheduling of these performance data is also important. For scheduling, we can use one of the scheduling managed object classes defined in X.746, Scheduling Function. Or, for simple implementations, NEs may just collect the performance data and transfer the performance data to the OS using FTAM.

3.10 Summary

This chapter covers management services and management functions. There are detailed discussions on the management services and management functions. It is necessary to have an explanation and feeling for the management functions as they form the base for systems management applications such as configuration management. This chapter ends with implementation details on how to develop management applications, taking performance management as an example.

3.11 References

3.1. ITU-T Recommendation M.3200, TMN Management Services and Telecommunications Managed Areas: Overview, 1997.

3.2. ITU-T Recommendation M.3400, TMN Management Functions, 1997.

3.3. ITU-T Recommendation M.3020, TMN Interface Specification Methodology, 1995.

3.4. ITU-T Recommendation Q822, Stage 1, Stage 2 and Stage 3 Description for the Q3 Interface—Performance Management, 1994.

3.5. ITU-T Recommendation Q821, Stage 2 and Stage 3 Description for the Q3 Interface—Alarm Surveillance, 1993.

3.6. ITU-T Recommendation Q823, Stage 2 and Stage 3 Functional Specifications for Traffic Management, 1996.

3.7. ITU-T Recommendation X.800, Security Architecture for Open Systems Interconnection for CCITT Applications, 1991.

3.8. ETSI GSM 12.00 (ETS 300 612-1), European Digital Cellular Telecommunication System (Phase 2), Network Management (NM): Part 1: Objectives and Structure of Network Management, 1996.

3.9. Udupa, D. K., *Network Management Systems Essentials,* New York: McGraw-Hill, 1996.

PART 2

TMN Information Model and Protocols

This page intentionally left blank.

CHAPTER 4

TMN Terms and Concepts

4.1 Introduction

Let us examine some of the important frequently used basic TMN concepts. As we alluded to in Chapter 1, many OSI systems management concepts, terms, and standards are also used in ITU-T TMN. As a result, we have the OSI prefix in many places. Some of the terms explained in this chapter, such as *polling* and *heartbeat*, are frequently used in TMN, systems management, and Internet network management.

This chapter will examine some important systems management concepts, such as management domains, management information hierarchies—the registration, inheritance, and containment hierarchies—object naming, scoping and filtering, synchronization, polymorphism, and allomorphism. Scoping and filtering will be useful in fault management. Also, we will discuss the topics of management state and the attributes used for relationships, which pertain to connecting managed objects. Management state and relationships between objects are very important concepts for configuration management. Though we have already identified fault management and configuration management, these terms will be used in other systems management functional areas as well. In addition, some of the terms will be used in discussing topics such as CMIP and managed object definitions.

We must be very careful in implementing ITU-T-related TMN standards. The standards documents provide broad guidelines but do not go into specific implementation details. Also, in some cases the definitions are in place but the details are still being worked out. Modifications may be made during implementation, and definitions and guidelines may be expanded to suit individual cases.

Also, in many cases the standard documents are in different stages of standardization. Standards should not be implemented while still in draft stage. Implementation has to be done when standards are fairly mature. Also, sometimes, to account for corrections and additions, amendments are made after standards are published. Therefore it is advisable to check whether any amendments to published standards are available during implementation.

The starting point in TMN and OSI systems management is the OSI seven-layer architecture. As per OSI seven-layer architecture, the different layers are the physical, data link, network, transport, session, presentation, and application layers. The topmost of these seven layers—the application layer—is important from the TMN and systems management point of view. TMN and systems management applications reside in the application layer.

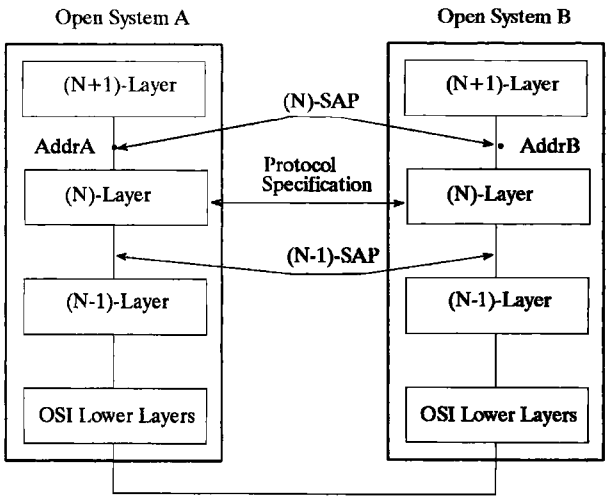


Figure 4-1
OSI layer concepts.

4.2 Service Access Point

In data communications and TMN, we often hear the term *entity*. Basically, an entity is an abstract concept (conceptual idea). An entity is that which provides services. Examples of services are error recovery, segmentation, and assembly. The entities in the (N) layer provide the services to the entities in the (N + 1) layer through a point known as the service access point (SAP). AddrA and AddrB are examples of SAP addresses.

The SAP is an addressing concept. A SAP is identified by a SAP address. It is a conceptual intersection between two layers. An (N + 1) entity may concurrently be attached to one or more SAPs attached to the same or different N entities and vice versa. So an entity in (N + 1) layer in open system A accesses the entity in the (N) layer using the intersection point address (AddrA) as shown in Figure 4-1. In a similar manner, an entity in the (N + 1) layer in open system B accesses the entity in the (N) layer using the SAP address AddrB.

4.3 Service Provider and Service User

An OSI service provider represents a collection of entities that provide an OSI service to an OSI service user. *OSI service user* refers to a single entity that makes use of an OSI service through an OSI service primitive as

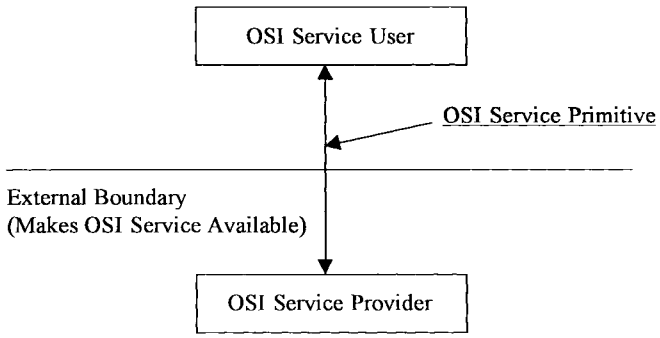


Figure 4-2
OSI service provider and OSI service user.

shown in Figure 4-2. OSI service can be defined as a capability provided by the OSI service provider and available for use by the OSI service user at the boundary demarcating OSI service provider and OSI service user.

OSI service primitive is an abstract concept; it represents an atomic implementation-dependent interaction between an OSI service provider and an OSI service user. Either the service provider or service user issues these service primitives. The service primitive contains information such as the semantics of the information carried in the service primitive, the constraints or conditions that must be met by the service provider or service user, and the actions to be performed on receiving the service primitive.

4.4 Service Definition and Protocol Specification

In ITU-T and ISO systems management documents, terms such as *service definition* and *protocol specification* are frequently used, so it is important to know what they mean. Some of the commonly used terms are:

- *Reference model:* The popular OSI seven-layer data communication architecture, used as the basis for the OSI standardization effort is an example of a reference model. Reference models provide the basic architecture for further standardization in an area. The B-ISD or Protocol Reference Model (Chapter 10) is another example of a reference model.
- *Service definition:* An abstract concept that includes the behavior of a service provider as seen by a service user. Alternatively, the service definition includes a set of capabilities provided to a service user by a

service provider. A service definition does not include the internal behavior of a service provider.

- *Protocol specification:* Furnishes a set of complete rules that govern the interaction between a service provider and a service user and provides implementation guidelines. As conformance requirements and testing relate to the implementation of protocols, they apply to protocol specification only. Most of the seven OSI layers are covered by service definition and protocol specification documents.

Concepts such as service provider, service user, service primitive, and service definition are explained in X.210, Conventions for the Definition of OSI Services (Reference 4.2).

4.5 Connection and Connectionless Modes

Some important discussions in TMN and OSI systems management are centered around connection (or *connection-oriented*) and connectionless modes of data communication. The basic difference between these two modes also contributes to differences in the protocols and how data communication is done.

To develop the concepts of connection and connectionless modes of data communication, let us first examine what a connection is. A cooperative relationship between two entities is known as an *association*. An identifiable association with an agreed-upon set of rules for data communication between two or more peer entities is known as a *connection*. Note that entities in the same layer are known as *peer entities*.

The *connection mode* of data transfer involves three distinct phases: connection establishment, the actual data transfer, and connection release. This is analogous to a telephone call. As a starting point for talking to someone, we need to form a telephone connection with the person we want to talk to. Then we talk, and after the conversation we release the connection by placing the receiver back on the hook. Of course, one or more telephone companies do the connection establishment, maintenance, and release. The connection mode of data transfer is useful when data transmission has to be done over a period of time, as in the case of file transfers, remote connection of a computer terminal to a computer, and so on.

In the *connectionless mode* of data transfer, no connection is established before data transfer between a source service access point and one or more destination service access points. Connectionless data transfer does

not have a distinguishable lifespan. All the required information, such as destination address, quality of service selection criteria, option, and so on, along with the unit of data to be transferred, are delivered to one or more service access points in a single service access. Notice that in the connectionless mode of data transfer, we do not employ the connection establishment and connection release phases used in connection mode of data transfer.

The connection mode, connectionless mode, and OSI seven-layer architecture are explained in X.200, Basic Reference Model: The Basic Model (Reference 4.1).

4.6 Service Primitives

Service primitives furnish further details on a state. There are four primitives associated with a connection mode: request, indication, response, and confirmation. When a request is sent, the sender may wait for a reply. In this case, the request is known as a *confirmed request.* In some cases the sender may not desire a reply to come from the responder. In such a case, the request is known as an *unconfirmed request.* A manager and an agent also use similar requests and responses, namely, request, indication, response, and confirmation. The behavior of these is shown in Figure 4-3. The commonly used generic terms *requester* and *responder* are used in this figure.

When a requester sends a message asking for a certain service to be performed by the responder, the requester sends it in the form of a *request.* On the responder side, correspondingly, the responder receives a message asking the responder to perform a service. This becomes an *indication* on

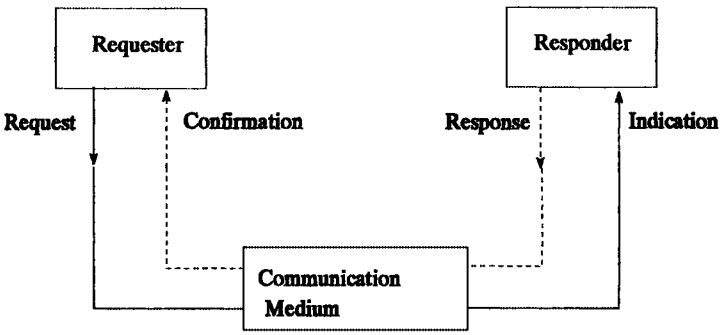


Figure 4-3
Requestor and responder messages.

the side of the responder. When a responder has to send some message, it sends it in the form of a *response*. On the receiving side, the requester sees the message as a *confirmation*. In the connectionless mode, only request and indication primitives are used. A requester sends a request primitive and a responder gets the indication primitive.

In the preceding explanations, the broad term *message* is used to avoid confusion that might arise if we referred to this communication with names such as *event report, notification, command,* or *response*. When specific terminology is required or clear distinctions are necessary, such terms will be well defined and explained.

Each service will have an identifier for the layer, for example, A, which refers to an application layer; a verb such as ASSOCIATE; and a primitive such as request. An example of a service is thus A-ASSOCIATE.request. Each service has parameters that convey information including data and controls.

4.7 Communication Between Managed Objects and Agents

Exchanges of messages between managers and agents are done by *protocol data units* (PDUs). PDUs contain data from the layer above and control information of a layer. A PDU can take the form of a request or response. A manager can send a request to an agent, and the agent in turn sends a response. These requests and responses contain management information, which is contained in parameters. However, it is not cast in concrete that only a manager sends requests and only an agent sends responses. In some cases, an agent may send unsolicited messages or notifications to the manager. The manager, in turn, may send responses to the notifications sent by the agent.

An agent must access managed objects frequently to know what is happening to them or how they are performing. One way to do this is through *polling*, in which the managed object is queried at constant time intervals. Polling is somewhat like asking the managed objects "How are you doing?" Sometimes polling is not that simple, however. For performance-oriented queries, managed objects may be required to send back large amounts of information. In polling, the time interval is an important factor. If polling is done too often, there is increased network traffic. On the other hand, if it is done at infrequent intervals, up-to-date information on the state of the

managed object may not be received. So the time interval used for polling is really a design decision that reflects the goals to be achieved.

In TMN, however, in case of unusual conditions, notifications are sent by the objects to an agent. These notifications, in turn, are converted to *event reports* in agents and sent to managers. This is a better way of learning the state of a managed object than polling. It avoids the unnecessary network traffic that results from polling. By and large, polling is unproductive and should be avoided.

Another way to access a managed object is to use another intermediary agent. This agent must be self-contained and is required to have intelligence to issue requests or responses, depending on the conditions observed in a managed object. Also, specific implementation-dependent methods can be used to access managed objects.

Sometimes a manager needs to know whether an agent is functional or "dead" due to some internal problem or a broken communication connection between a manager and an agent. In such a case, the manager uses *heartbeats*, whereby an agent sends a message to the manager at regular intervals saying "I am alive." Like polling, the heartbeat may also carry information on the state of an agent.

4.8 Management Domain

Management domain is an important concept that helps in distributing management functions. By carefully designing management domains, we are able to conveniently partition systems management into manageable portions. There may be some gateways, different computer systems belonging to different vendors, and different kinds of systems, from workstations to mainframes. How do we show these on the screen of a workstation? When we show topology, it must be of some use and make sense. One intuitive way is to break up the topology into easily manageable management domains. Imagine the case of a network that spans a large geographical area, connecting some continents. In this example, each nation could form one management domain.

A collection of managed objects for systems management and TMN purposes is known as a management domain. The division of management domains may be based on geography, functions, or technology, and it helps in the application of management policies to a group of managed objects. We show one example of the concept of management domain in Figure 4-4.

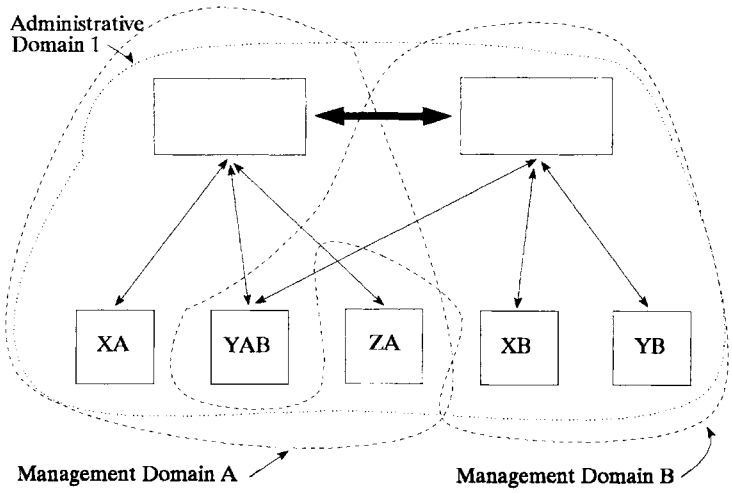


Figure 4-4
Management and administrative domains.

Management domains, to be useful, must have unique names, must include the objects that will be managed by that domain, and must know how the managed objects and agents will communicate with one another. One managed object in one management domain may also belong to another management domain, as shown in Figure 4-4. From Figure 4-4, notice that management domain A has managed objects XA, YAB, and ZA as members. Similarly, management domain B has managed objects YAB, XB, and YB as members. Thus, it is apparent that managed object YAB is a member of both management domains A and B.

After the whole network has been divided into management domains, what do we do? If there is no centralized control, there may be chaos. For this reason, we define another set of domains over this conceptual model. This is known as the management *administrative domain*, as shown in Figure 4-4. The functions of management administrative domains are as follows:

- Exercise control over managed objects and agents in the domain.
- Aid in changing the boundaries of management domains if, for example, after some time, certain managed objects need to be under the control of another management domain.
- Facilitate coordination when a managed object belongs to more than one management domain.

X.749 (ISO 10164-19), Management Domain and Management Policy Management Functions (Reference 4.7), provides information on managed object classes for management domains and management policies,

models for the behavior of management domains and management policy, the services that can be performed, mapping of the services to CMISE services, and a model for retrieving information on the managed objects associated with management domains. Managed object classes defined for management domains are shown in Figure 4-5. The explanations of these managed object classes are as follows:

- *domainCoordinator* provides an interface to a management domain for management operations. domainCoordinator represents the functional aspects of a management domain, such as the members of the domain, the relationship with a managementPolicy managed object, and the management policy as applied to members of a domain. domainCoordinator can coordinate domains, superdomains, and subdomains. A domain within another management domain is known as a *subdomain*. The parent management domain is known as the *superdomain*.
- *managementPolicy* is responsible for the management policies that can be applied to the members of a management domain. This managed object class contains a list of the authorities allowed to access and modify the management policies.
- *conflictDetector* detects conflicts in management policies of managed objects in a management domain or domains.

Some of the management operations that can be performed on a management domain are subdomainCreate, domainDelete, enrollMember, de-enrollMember, changeLocalName, listParents, listMembers, and listSubdomains. These management operations are used as action-type parameters in CMIS M-ACTION (M-ACTION will be discussed in Chapter 5.) Similarly, the action types associated with management policies are policyCreate, policyDelete, policyScope, enrollRule, and de-enrollRule.

A management domain is created by a manager or an administration by creating an instance of a management relationship among management policy, a member of a domain, and a domain coordinator. In management domains, there are still some unresolved issues, including:

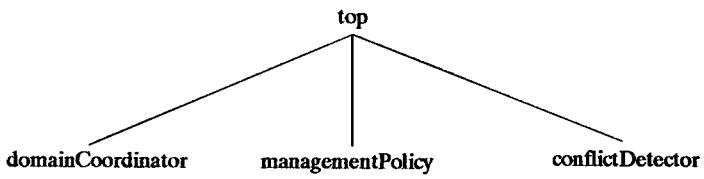


Figure 4-5
Managed object classes for management domains.

- Security issues involved in accessing other domains and managed objects in other domains.
- The strategy for backup when one domain fails. Do we allow one domain to be the backup for another domain? The ISO 10164-19 document does not address this issue.
- X.701, Systems Management Overview (Reference 4.3) mentions the administrative domains. For implementation purposes, administrative domains must be formally defined.

4.9 Management Information Hierarchies

Managed objects have relationships with one another. These relationships can be organized in a systematic manner by indicating how one managed object is related to another object. These relationships can serve different purposes. If managed objects are to be globally known to all, then they must have relationships indicating where they stand in a global naming structure. For this purpose, a *registration hierarchy* is used.

Because OSI management standards use object-oriented principles, it may be possible to use some characteristics already defined for other objects. This is done through *inheritance*. This relationship between managed object classes is provided by an *inheritance hierarchy*.

We may wish to have a relationship for naming a managed object. This relationship is shown by a *containment hierarchy*, also sometimes known as a *naming hierarchy*. These three hierarchies that are used for management purposes are shown in Figure 4-6. It should be noted that these hierarchies are independent of each other and are used for different purposes.

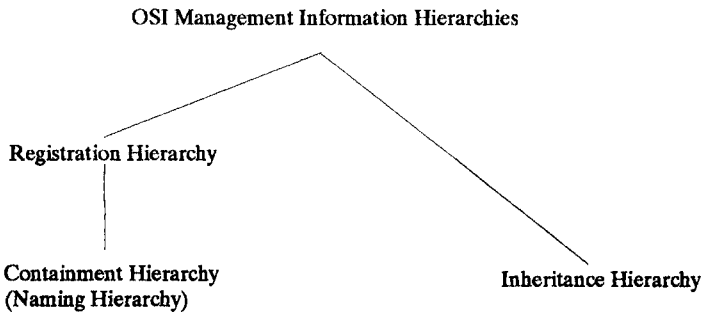


Figure 4-6
OSI management information hierarchies.

4.9.1 Registration Hierarchy

A managed object class is identified by an ASN.1 object identifier. These object identifiers are formed by taking a sequence of integers in the registration tree. These numbers are registered; the registration hierarchy tree is shown in Figure 4-7. The registration number for the Internet is {1.3.6.1}. All systems management object identifiers, which are used in standard documents, are derived from {joint-iso-itu-t ms(9)}.

These numbers, from the root to the leaf of the registration hierarchy tree, are concatenated to form an *object identifier.* An object identifier is a series of integers used to uniquely identify a managed object class. We should note here that an attribute also has an object identifier to uniquely identify it.

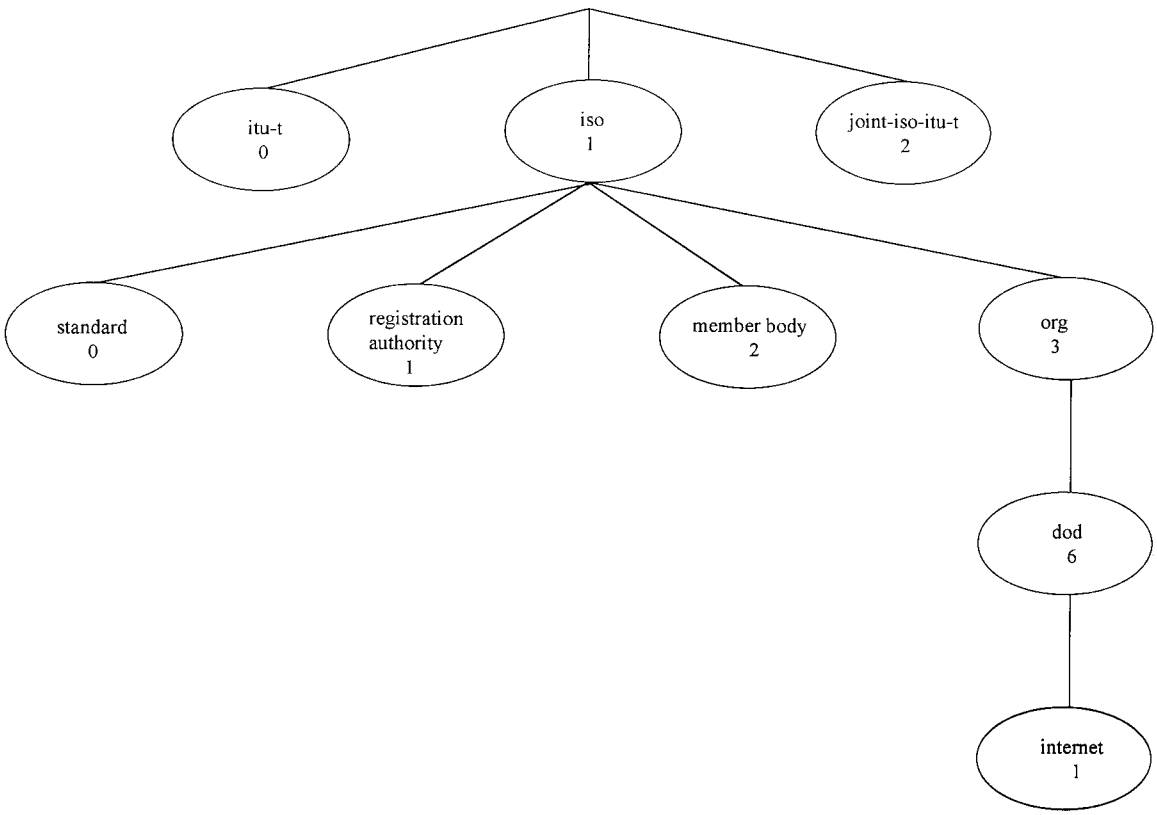


Figure 4-7
Registration hierarchy.

4.9.2 Inheritance Hierarchy

It is important to know where a managed object class is positioned in the inheritance hierarchy tree. The property of inheritance is very useful, because we can derive characteristics from parents. In other words, it is not necessary to define the characteristics that parents already have. Parents, in turn, inherit properties from their parents; thus, a new managed object class definition reduces to properly positioning a managed object class in the inheritance hierarchy tree and adding some more characteristics.

Inheritance hierarchy is derived from the managed object class *top*. From *top*, various organizations and countries are derived. An example of an inheritance hierarchy tree is shown in Figure 4-8. Again, a managed object class may be derived from one or more managed object classes, which facilitates inheriting characteristics from those managed object classes. We call this *multiple inheritance*. Figure 4-8 also illustrates multiple inheritance.

The managed object class *lanNetworks*, which is the parent of another managed object class *lanNet*, is known as a superclass with respect to the managed object class *lanNet*. Similarly, a managed object class *lanNet*, which is the child of managed object class *lanNetworks*, is known as a subclass.

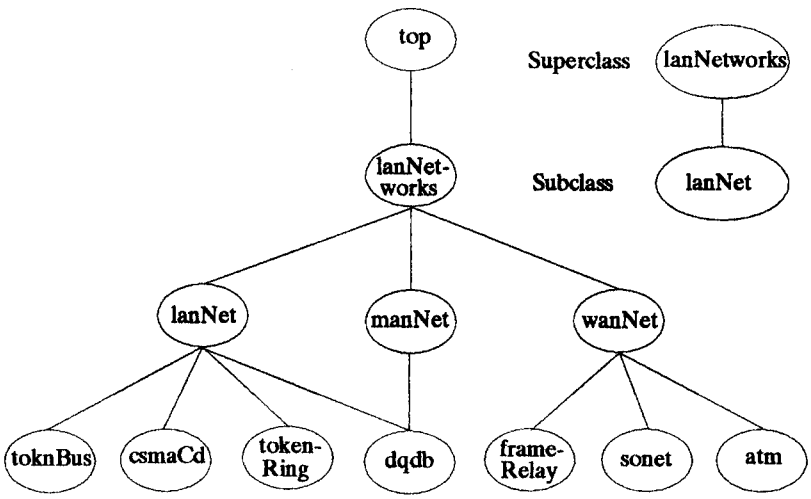


Figure 4-8
Inheritance hierarchy.

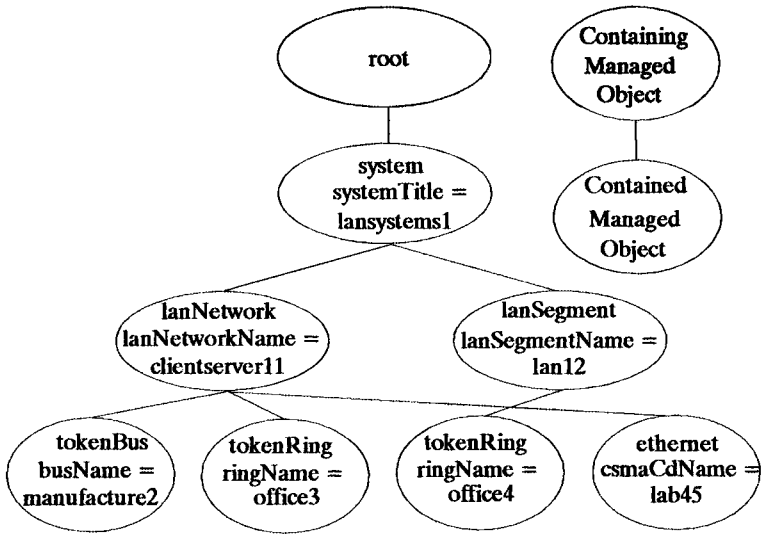


Figure 4-9 Containment hierarchy example.

4.9.3 Containment Hierarchy

One managed object can represent a part of another managed object. This relationship is commonly known as *containment* and the structure is used for constructing global names. This containment relationship is shown in Figure 4-9. Here, system *lansystems1* is known as a contained managed object in relation to *root*. And *root*, in turn, is known as a *containing object*.

The object instance of *lanNetwork* named *clientserver11* is a contained managed object with respect to the object instance of the system named *lansystems1*. The managed object *lansystems1* is the containing managed object.

The containment relationship is relevant to object instances. We must remember that it does not apply to managed object classes. Containment is a directed graph where arcs connect contained and containing objects. Containment relationships can refer to static and dynamic behaviors that must be specified when defining containment relationships.

4.10 Object Naming

We use names to recognize people. Similarly, we use a name to identify a managed object. Containment and naming are closely related to one another. A containment tree is used for naming a managed object. Every

managed object must have a unique name. The containment hierarchy fans out from *root*. In the containment hierarchy, *root* is the starting point and represents a null object. Containment relationships can be described in the definitions of a managed object class, but are often described after classes are defined.

Referring again to Figure 4-9, some more terms used in defining a managed object class need to be explained. Here, the object instance of *lanNetwork* is a subordinate object in relation to the object instance of the managed object class *system*. The object instance of *system* is the superior object in relation to the object *lanNetwork*.

If we extend this relationship, *root* is the superior object class of *system*. This hierarchy of relationships forms a tree known as the *naming tree*. It is important to mention how we name a managed object of a class that we are defining. The relationship of a subordinate object class to the superior object class is known as *name binding*. In our discussions, *naming tree* and *containment tree* are used interchangeably.

Naming is important for uniquely identifying an object instance. In the containment tree, the distinguishing attribute, along with the name of an object instance, is known as the *relative distinguished name* (RDN). Referring to Figure 4-9, the RDN for *tokenRing* is {ringName=office3}. The RDN need not be unique at the same level; however, in relation to a superior managed object, the RDN must be unique.

The distinguishing attribute used for uniquely identifying a managed object is part of the mandatory package and, obviously, should have a fixed value throughout the life of the managed object. However, there need not be just one distinguishing attribute; there can be multiple attributes used for different name bindings.

In Figure 4-9, how do we distinguish between object instances of *tokenRing* with the RDNs *ringName=office3* and *ringName=office4* under the managed object classes *lanNetwork* and *lanSegment?* To uniquely identify the *tokenRing* object instances, we must move up the containment tree. For this we need the help of the RDN of the parents. Starting from the beginning of the containment tree, if we concatenate the RDNs, we can derive the globally unique names. A globally unique name is known as a *distinguished name* (DN), and it is unique across the entire containment tree. The DN for *tokenRing* is {systemTitle=lansystems1, lanNetwork-Name=clientserver11, ringName=office3}. It is important to note that the containment tree and the physical containment of one resource by another need not be similar.

For naming a systems managed object, we use systemID or systemTitle. The ASN.1 for systemID can be a GraphicString, Integer, or Null. A Null value is used when a system has not been configured or a systemID

attribute is not used in naming. A system title is a layer-independent name. In systemTitle, the ASN.1 type can be a distinguished name, Object Identifier, or Null.

There are two forms of naming for systems management purposes: local or global. For global naming, we start from the root in the containment tree and proceed down the naming tree, concatenating the RDN until we arrive at the managed object desired. In contrast, for the local name of a managed object, we can start from any managed object in the naming tree. However, for OSI systems management, we take the systems managed object as a starting point for local names. The local name of a systems managed object is an empty sequence shown by {}. A local name may not be globally unique.

Let us examine this with the example shown in Figure 4-9. The global name for *tokenRing* is {systemTitle=lansystems1, lanNetworkName= clientserver11, ringName=office3}. In global naming, we use *root* as the reference point. Global names cannot be used for a system when the systemID and systemTitle are both Null, because this would show that the system is unaware of its global name. In this case, the local name is {lanNetworkName=clientserver11, ringName=office3}. Because a systems managed object is represented by {}, {} is not included in the local name.

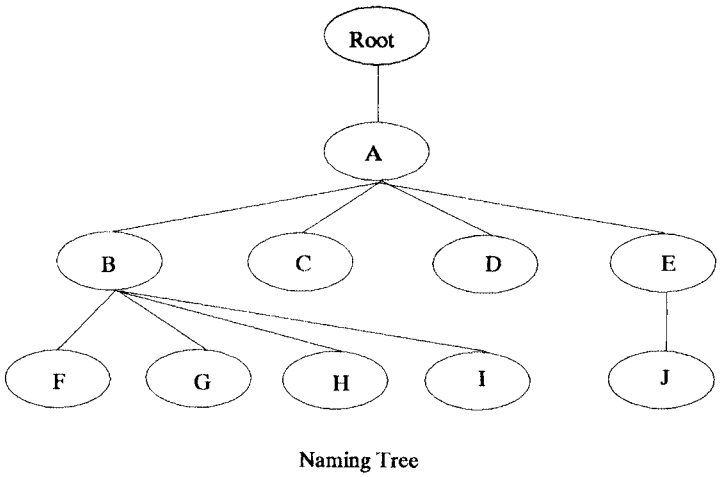
4.11 Scoping

When management operations are done on managed objects, *scoping* enters into the picture. These management operations are provided by CMISE services. Scoping selects one or more managed objects for management operations such as Action, Delete, Get, and Set requests. The scoping is applied to a naming tree. The base object can be anywhere in a naming tree, but, to be successful, the request should be sent to the agent with the base object. The base object specified in the request is the reference point for scoping operations. Base object level is referred to as level zero, and sometimes the subtree below a base object is used in scoping.

There are different levels of scoping, as follows:

1. A base object alone is selected. This is the default action for scoping. In Figure 4-10, if the argument of a scoping operation is the base object only and the base object specified in a request is A, then the result is a selection of A. Note that the base object is A.
2. The *n*th-level subordinates can be selected. For example, if *n* is the first level in Figure 4-10, the objects selected are B, C, D, and E.

Figure 4-10
Scoping example.



3. Scoping can be applied to objects that include the base object and the subordinates up to the *n*th level. In Figure 4-10, take *n* as one; then managed object A will be selected in addition to B, C, D, and E.
4. Scoping can be extended to cover all objects in the subtree below the base object. If we take A as the base object, the selected objects from this scoped operation, are A, the objects in the first level (B, C, D, and E), and the objects in the second level (F, G, H, I, and J).

4.12 Filtering

Filtering is one more level of selection imposed on scoping. It can be used to choose a subset of managed objects that have been selected by the scoping operation. Those managed objects that have been chosen by scoping are further subjected to conditions furnished by filtering for the selection. If the managed objects pass the conditions, they are selected; otherwise, they are left out. These conditions are formed by grouping logical operations such as *or, and,* and *not.*

When all conditions are true, the *and* condition is satisfied and becomes true. In the case of the *or* condition, if one or more conditions are true, then the whole condition set is true. In the case of *not,* the condition is true only if the nested filter (a combination of filters) is false.

For evaluating whether an attribute value matches a certain rule, there are matching rules. The first condition is that the attribute must be

present. If the attribute is not present, then the comparing value is regarded as false. There are eight explicit matching rules:

1. *Equality:* We can test an attribute value to equality with an assigned value. As an example, in Figure 4-11, we can test whether *tokenRingCardPrice* has a value of 300. In this case, we get the object instance D. An attribute may have more than one value, in which case we get a set of values. We can also use a set for comparison. This set also may have one or more members. For a matching rule to become true, members of the comparison set must be equal in value to all members of the attribute set. Otherwise, the matching rule evaluates as false.
2. *Greater than or equal to:* Here, the value supplied for comparison must be equal to or greater than the attribute value. For example, in Figure 4-11, if we take a comparison value of 300, we are looking for object instance values for the attribute *tokenRingCardPrice* of 300 and less. The object instance is D. For sets, the comparison set has only one member. For the matching rule to become true, the value of the member of the comparison set must be greater than or equal to one or more members of the attribute set, or else the matching rule evaluates as false.
3. *Less than or equal to:* In this test, the value used for comparison must be less than or equal to the attribute value. For comparison, let the value be 325. From Figure 4-11, for *tokenRingCardPrice*, the object instances with values greater than or equal to 325 are C and E.

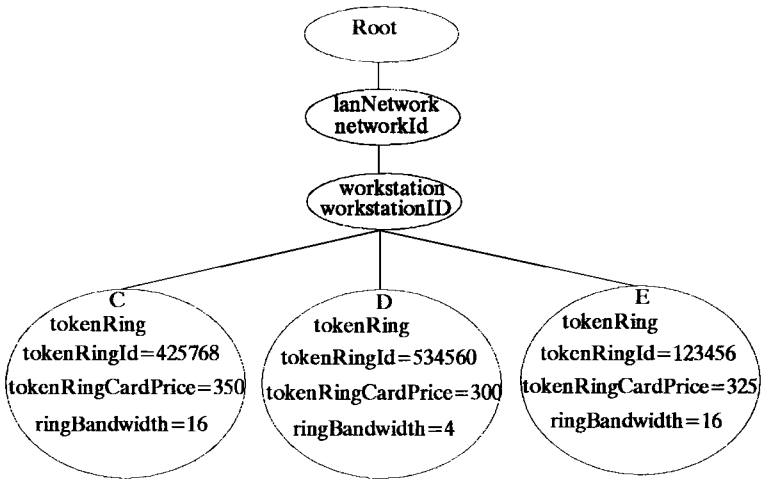


Figure 4-11
Filtering example.

4. *Present:* This tests whether an attribute is present, and if so, then the comparison is evaluated as true. The attribute selected may be *collisionRate*. A cursory look at the *tokenRing* managed object class in Figure 4-11 reveals that there is no such attribute. In this case, there are no object instances that can be selected.
5. *Substring:* For this test, the attribute value substring must match the substring furnished for comparison. Let us check in Figure 4-11 for an object instance that has a *tokenRingId* of 123456. On comparing each element of the substring, the object instance of E is obtained.
6. *Subset of:* Here the set for comparison must be a subset of the attribute values. As an example, let the set of *tokenRingId* for comparison be {123456, 325}. On examination of the naming tree in Figure 4-11, only E matches. *Subset of* is applicable only to set-valued attributes.
7. *Superset of:* This test is applicable only to set-valued attributes, and all members must be present in the set presented for comparison. Referring to Figure 4-11 again, for *tokenRingId*, let the set for comparison be {123456, 325, 987234}. On comparison, there is no object instance matching this requirement.
8. *Non-null set intersection:* Again in Figure 4-11, let the set for comparison be {123456, 456789, 987234}. For this to be evaluated as true, there must be at least one *tokenRingId* with one of the above values. In our example, the object instance is again E.

We can combine these simple matching rules and evaluate them as true or false. For example, let the combination of filters used be (*objectClass=tokenRing*) *and* (*tokenRingCardPrice*=350). The object instance is C only. For variation, let the combination of filters be *tokenRingCardPrice*=300) *or* (*ringBandwidth*=16). Now the object instances are C, D, and E.

4.13 Synchronization

A manager may request information from an agent about how the agent performs operations on behalf of the manager. This is known as *synchronization*. Sometimes operations are performed on more than one managed object. Synchronization is part of managed object behavior definition. We will examine what behavior means in relation to the definition of a managed object in Chapter 6, Section 6.4.5.

Synchronization does not apply to a Create operation on an object. Here, intermediate operations are not visible. When operations on objects such as Get, Set, and Action include scoping, the issue of synchronization arises. The operations that can be performed on managed objects are explained in Chapter 5. The synchronization operation can be of two types. The first, *atomic* synchronization, is an all-or-nothing proposition. An initial check is made as to whether it is possible to satisfy the request for all objects. If it is not possible to retrieve all the objects, say for a Get operation, then an error is returned. Otherwise, all the objects are returned. The second type of synchronization is known as *best-effort* synchronization. For example, for a Get operation, all retrievals are tried and whichever objects can be retrieved are returned. For those cases where retrieval fails, an error is returned.

4.14 Polymorphism

Polymorphism means *having many forms* and is primarily an object-oriented concept. Managed object classes that respond to a common operation in a similar manner are said to exhibit the property of polymorphism. Referring to Figure 4-12, let us assume that we have many managed object classes in an undergraduate level based on a student's major. Thus, there are managed object classes such as *computerScience-*

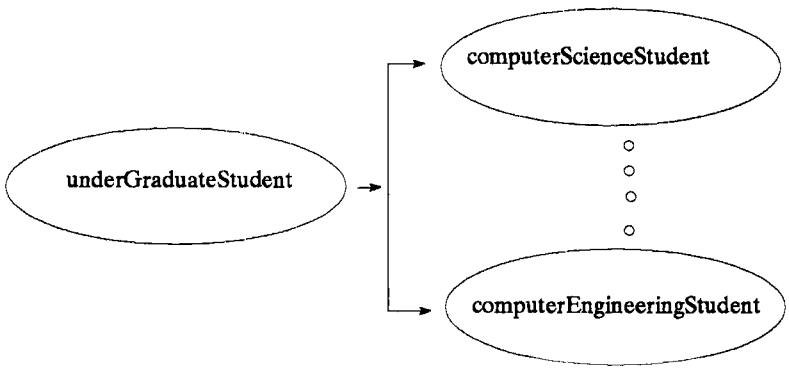


Figure 4-12
Polymorphism example.

Student, computerEngineeringStudent, and so on. Suppose we have an operation such as Grading for the managed object class *underGraduateStudent.* As a result of the property of inheritance, this operation of Grading will be inherited by *computerScienceStudent* and *computerEngineeringStudent,* and may need only slight modifications for the subclasses of *computerScienceStudent* and *electricalEngineeringStudent.* Here the managed object classes *computerScienceStudent* and *computerEngineeringStudent* are termed *polymorphic.*

4.15 Allomorphism

Allomorphism refers to the ability of one object instance to function as if it were part of more than one object class. To understand this concept, let us slightly modify our earlier example. Assume that a student, Joe Smith, is majoring in computer science. Further assume that we have the managed object classes *xUniversityStudent, underGraduateStudent,* and *computerScienceStudent. Joe Smith* can be taken as an instance of the managed object class *computerScienceStudent.* However, the object instance *Joe Smith* can also be part of the managed object classes *underGraduateStudent* and *xUniversityStudent.* Here, *Joe Smith* has the characteristics of allomorphism. Allomorphism is used in OSI systems management and TMN to migrate to newer versions of the old or obsolete objects.

4.16 Management State

State describes operational and availability conditions of a managed object at a specific time. These conditions are included when appropriate in the definition of a managed object class as attributes. It is necessary to standardize the state for management purposes.

Management state attributes are broadly classified as *generic state* attributes and *status* attributes. The status attributes furnish further information on generic state attributes. The generic state attributes are operational, usage, and administrative, as shown in Figure 4-13. The status attributes used for management purposes are *alarm, procedural, availability, control, standby,* and *unknown.* Management state attributes and their values are listed in Table 4-1.

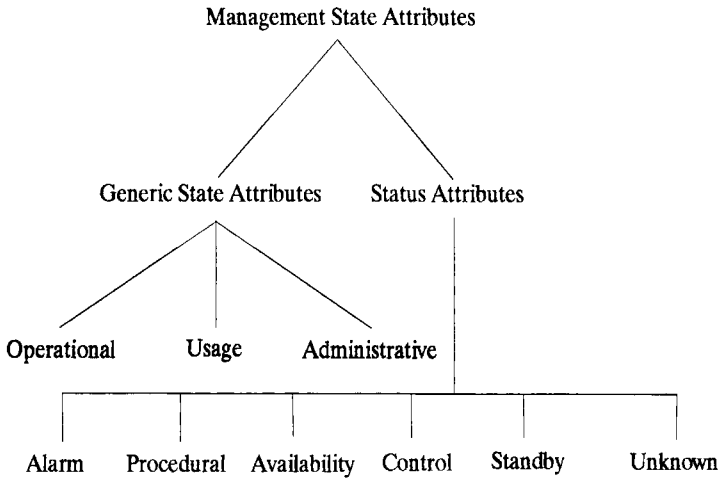


Figure 4-13
Management state attributes.

4.16.1 Generic State Attributes

The *operational* state attribute indicates whether a resource is working. Management operation can only read the operational state of a managed object; hence, this state has only a Read operation and is single-valued. There are two values for this attribute: *disabled* and *enabled*. When the operational state is not known, then the *unknown* status attribute with a value of *true* is used to reflect this condition.

TABLE 4-1
Different Management State Attributes.

State Attributes	Possible States
Generic State Attributes	
Operational	Disabled, enabled
Usage	Idle, active, busy
Administrative	Locked, unlocked, shutting down
Status Attributes	
Alarm	Under repair, critical, major, minor, alarm outstanding
Procedural	Initialization required, not initialized, initializing, reporting, terminating
Availability	In test, failed, power off, off line, off duty, dependency, degraded, not installed, log full
Control	Subject to test, part of services locked, reserved for test, suspended
Standby	Hot standby, cold standby, providing service
Unknown	Used when state of a resource is unknown

The *usage* state defines whether a resource is being actively used at a particular time. It mentions whether additional users can use this resource. There are three values: *idle, active,* and *busy.* The usage state attribute is also a read-only and single-valued attribute. When only one user is involved, the possible values are *idle* or *busy.* When a resource has an unlimited number of users, then the possible values are *idle* or *active.* As in the operational state, when resource state details cannot be furnished by the usage state, the unknown status attribute with a value of *true* is used.

The *administrative* state indicates how a resource is used under management operations. The possible values are *locked, unlocked,* or *shutting down.* Some managed object classes have only a subset of the three values. In some cases, there is no unlocked state. In some other cases, the *shutting down* value is absent, if there is no graceful shutdown characteristic. Administrative state values are stated while defining managed objects. Note also that the administrative state attribute is read-write and single-valued. That means that the value of an attribute can be read by using a Get operation and changed by doing a Set operation.

When defining a managed object class, it is possible to have one or more generic state attributes. However, when defining values, it must be ascertained that these values represent only a valid combination of possible values.

4.16.2 Status Attributes

Status attributes furnish further details on the generic state attributes, such as *operational, usage,* or *administrative.* Let us examine these status attributes individually.

The *alarm* status attribute indicates the status of alarms that have been generated. It can have a value of *under repair,* which states that the resource is being repaired, or *critical, major,* and *minor,* which indicate that one or more alarms have been detected and have not been cleared. In all of these cases, including the *under repair* value, the operational state can be enabled or disabled. The last possible value, *alarm outstanding,* states that one or more alarms are outstanding against the resource and the condition due to these alarms may or may not have been disabled. It is not necessary that the alarm status attribute have values. Instead, this attribute may be an empty set, which indicates that none of the aforementioned values are present. The alarm status attribute is read-write, indicating that Get and Set operations can be performed.

The *procedural* status attribute is used when a resource has many phases of operations. If this value attribute is an empty set, then the managed object representing the resource is *ready.* One of the attributes is *initialization required,* which indicates that this managed object has to be initialized before it can perform its normal role, and the operational state is disabled. On the contrary, a value of *not initialized* indicates that the managed object is capable of initializing itself and needs no initialization to perform its normal functioning; however, initialization has not been started. The operational state here can be enabled or disabled. The value of *initializing* indicates that the initialization process is in progress, and the operational state may be disabled or enabled, depending on the definition of the managed object class. *Reporting* states that the object is reporting the results of the operation that has been done, and the operational state is enabled. As the name suggests, the *terminating* value states that the resource is in a termination phase. Again, it is not necessary to have one of these values for an attribute; the value of an attribute can be an empty set.

The *availability* status attribute indicates whether a managed object is available. It has values of *in test, failed, power off, off line, off duty, dependency, degraded, not installed,* or *log full.* Here, *off line* indicates that the operational state is disabled and needs some sort of intervention to make it available for use. *Dependency* states that some dependent resource is unavailable and the operational state is disabled. Other values are obvious, so the meanings are not discussed. Availability status is a read-only attribute. It is worth noting that the alarm, procedural, and availability status attributes add further meaning to the operational state attribute. As mentioned earlier regarding other status attributes, this attribute can be an empty set.

The *control* status attribute is read and write, and the values are *subject to test, part of services locked, reserved for test,* or *suspended.* The value of *subject to test* indicates that the resource may be subjected to tests during normal operation, and it may sometimes lead to abnormal behavior. *Part of services locked* means that some services of a resource may not be available for users. *Reserved for test* states that a resource is not available for normal users and is undergoing a test. *Suspended* indicates that a resource is not available for users.

The *standby* status attribute states the condition of a backup resource. This is a read-only attribute. The value of *hot standby* means that the standby resource does not need initialization and contains all the information about the resource the standby resource is backing up; thus, whenever a resource needs the backup to take over, the backup will be

immediately available. However, in the case of *cold standby,* the backup resource needs some sort of initialization before it can take over from a resource. The *providing service* value states that backup operation is already being done by the backup resource. *Unknown status* is used when the state of a resource is not known.

In addition to the state and status attributes used when defining a managed object class, there is also an attribute group of *state,* which represents a collection of all the state attributes. Using the state attribute group makes it easier to perform management operations on the collection, because this avoids the need for performing the same management operation on each individual state attribute. In Chapter 6, Section 6.4.3, we will examine in detail the meaning of attribute groups.

Whenever there are changes in the management state attributes of a managed object, an agent will send an M-EVENT-REPORT to the manager. This M-EVENT-REPORT can be a confirmed or nonconfirmed service. The parameters in the M-EVENT-REPORT for reporting management state changes are *event type, event information,* and *event reply.* Here, we must be clear that the management state includes generic states and status.

Let us briefly look into the parameters and the values that go in these parameters when using M-EVENT-REPORT to report management state changes. Event type is used to report a change in the value of a state attribute. Event information furnishes details of the event due to a change in state and has a source indicator with three values: *resource operation, management operation,* and *unknown.* Event information also has an *attribute identifier list* and a *state change definition.* The state change definition has an *attribute identifier, old attribute value,* and *new attribute value.* There is no event reply parameter specified for the state change notification. In addition, the parameters used with M-EVENT-REPORT, such as notification identifier, correlated notification, additional text, and additional information, can also be used to furnish further details on state change. We will revisit M-EVENT-REPORT, along with the parameters used, in Chapter 7. State and status attributes are explained in X.731, State Management Function (Reference 4.5).

4.17 Attributes for Relationships

A system may consist of many managed objects. If so, how these objects are related or grouped becomes important. It is also important to know how relationships change when certain operations are done. For this, the

relationship attribute *role* is used. The role attribute may have a single value or a set of values. A managed object class such as *printerUsed* may have another backup managed object class such as *printerBackup*. How these managed object classes are related and how the backup managed object class *printerBackup* behaves when the original managed object class *printerUsed* fails is important. Different kinds of relationships are explained in X.732, Attributes for Representing Relationships (Reference 4.6). When a relationship changes, notification is produced. This notification can be sent as an M-EVENT-REPORT. An agent sends M-EVENT-REPORTs to a manager. Parameters of an M-EVENT-REPORT are tailored to identify managed objects participating in a relationship and changes in the relationship. Details of the parameters of M-EVENT-REPORT for reporting relationship changes can be found in X.732, Attributes for Representing Relationships (Reference 4.6).

4.18 General Relationship Model

X.725, General Relationship Model (Reference 4.4), primarily describes how to represent relationships between resources and the management operations that can be done on relationships. Managed objects can be tied to one another for management purposes by managed relationships, and the managed relationships that share the same definition can be grouped as a *managed relationship class*. A manager uses the managed relationships, and these managed relationships are useful for different systems management functions, especially for configuration management.

Management operations that can be performed on a managed relationship are BIND, Establish, Notify, Query, Terminate, Unbind, and User DEFINED. The meanings of some of the management operations are obvious, so we will discuss only certain ones. BIND refers to associating a managed object with a relationship. ESTABLISH stands for forming a managed relationship. NOTIFY is used to report on events associated with managed relationships. USER DEFINED, as the name suggests, can be employed to carry a user-defined management operation. These management operations are mapped to systems management operations such as attribute-based and managed object-based management operations.

For relationships, *relationshipObjectSuperClass* under *top* is defined. This managed object class contains the relationship attribute group and the following attributes:

- *Relationship name:* Used for naming relationship objects
- *Relationship class:* Identifies the relationship class
- *Role binding:* Specifies how a managed relationship is represented

We have briefly discussed the general relationship model; for further details, readers should consult X.725, General Relationship Model (Reference 4.4).

4.19 Management Information Tree (MIT)

Managed object instances are arranged in a hierarchical fashion, forming a tree. This hierarchical tree forms the containment hierarchy for naming the managed objects instances and is known as a management information tree (MIT) (see Figure 4-9). An MIT is dynamic and changes with the deletions or additions of managed object instances. Management information is stored in the nodes of the MIT, including the internal and leaf nodes. Agents interface with the object instances in the MIT.

Normally, when an agent starts up, it creates managed object instances in an MIT. The number of object instances in the MIT can sometimes be very large in real-life situations. So the managed object instances can be arranged in the form of one of the variations of B trees for faster accesses and searches. To reduce access and search times, the object instances can also be arranged in the form of distributed trees.

The details of the managed objects, which are required for management purposes, are stored in a conceptual repository known as a management information base (MIB). A MIB is a conceptual model and it has no relation to how the data is physically or logically formatted and stored. MIB structure does not change dynamically like a MIT does.

4.20 Intelligent Agents

Not many functions are handled by agents. Agents control managed objects, send notifications to managers on unusual occurrences in agents and managed objects, and send responses to the commands from managers to management operations. Otherwise, most of the management

functions reside in managers. In SNMP protocols, for extensibility reasons, the bare minimum functions and intelligence reside in agents.

This is good from one angle: It is easy to add new equipment and hence new MIBs into the management structure. At the same time, the centralized network management imposes a heavy workload on the managers. To overcome this, the distributed network management is also becoming increasingly popular. We will look into more on distributed network management in Chapter 11.

We briefly present the ongoing work and different approaches taken to incorporate more intelligence in agents. The motivation for this brief overview is to acquaint readers with some of the possible directions and solutions to divert the workload from the managers. This becomes a very critical issue as networks are becoming global and heterogeneous, and the mix of data carried on the networks is more complex than a few years earlier.

To reduce the workload on the managers, different approaches are taken. One of the solutions is to use distributed network management, but others are being suggested. One of the approaches involves adding more intelligence and processing in agents themselves by using decentralized management by delegation (Reference 4.8). This reduces the network bandwidth required for transferring network management data and reduces the processing needs of the managers. In management by delegation, a flexible elastic process in an agent is used. An elastic process can perform additional processing off-loaded by other processes. The management applications reside in agents and a good amount of processing is done in the elastic process. So, more intelligence is added to the devices or agents, reducing the management functionality required of managers.

Another approach taken by the authors of Reference 4.9 is related to service management. Here service management activities are separated from the network layer activities. The service management semantics, as well as data and logic to interpret data, are incorporated in service agents. These service agents are mobile in the sense that they are location independent.

One more idea is have an intermediate layer between a manager and agents. The intermediate layer has managing agents that act as managers to the agents below and perform like agents to the manager above. This concept shifts the workload from the manager to the managing agent. In the work cited in Reference 4.10, the managing agents have intelligence and perform functions as a rule-based expert system.

While different approaches are being tried, it is important to note that there are neither standards for the intelligent agents as such nor popular and well-accepted implementations.

4.21 Summary

In this chapter, we have discussed some of the important concepts in TMN such as service access points, service providers, service users, service definitions, and protocol specifications. We have also introduced the topics of connection and connectionless modes of data transfer. Connection and connectionless modes of data transfer are important concepts and are used in CMIP and SNMP management protocols.

We have also discussed the concepts of management domain and management information hierarchies. We have examined how managed objects and agents communicate. While discussing management information hierarchies, we have examined three different types of hierarchies: registration, inheritance, and containment.

The important concept of naming a managed object has been examined in detail. Scoping and filtering have also been discussed with examples, as well as synchronization, polymorphism, allomorphism, management state attributes, and the role attributes that govern the relationship between different managed objects. We have ended this chapter with a brief discussion on management information tree and intelligent agents.

4.22 References

4.1. ITU-T Recommendation X.200 (ISO 7498-1), Information Technology, Open Systems Interconnection, Basic Reference Model, The Basic Model, 1994.

4.2. ITU-T Recommendation X.210 (ISO 10731), Information Technology, Open Systems Interconnection, Basic Reference Model, Conventions for the Definition of OSI Services, 1993.

4.3. ITU-T Recommendation X.701 (ISO IS 10040), Information Technology, Open Systems Interconnection, Systems Management Overview, 1992.

4.4. ITU-T Recommendation X.725 (ISO 10165-7), Information Technology, Open Systems Interconnection, Structure of Management Information: General Relationship Model, 1995.

4.5. ITU-T Recommendation X.731 (ISO 10164-2), Information Technology, Open Systems Interconnection, Systems Management: State Management Function, 1992.

4.6. ITU-T Recommendation X.732 (ISO 10164-3), Information Technology, Open Systems Interconnection, Systems Management: Attributes for Representing Relationships, 1992.

4.7. ITU-T Recommendation X.749 (ISO DIS 10164-19), Information Technology, Open Systems Interconnection, Systems Management, Part 19: Management Domain and Management Policy Management Functions.

4.8. Meyer, K., M. Erlinger, J. Betser, C. Sunshine, G. Goldszmidt, and Y. Yemini, *Decentralizing Control and Intelligence in Network Management.* Integrated Network Management IV, Proceedings of the Fourth International Symposium on Integrated Network Management, 1995. Sethi, A. S., Raynaud, Y. and Faure-Vincent, F. (eds.), London: Chapman & Hall, 1995, pp. 4—15.

4.9. Hjalmtysson, G. and A. Jain, *An Agent-based Approach to Service Management—Towards Service Independent Network Architecture.* Integrated Network Management V, Integrated Management in a Virtual World, Proceedings of the Fifth IFIP/IEEE International Symposium on Integrated Network Management, San Diego, CA, May 12—16, 1997, Lazar, A., Saracco, R., and Stadler, R. (eds.), London: Chapman & Hall, 1997, pp. 715—729.

4.10. Trommer, M. and R. Konopka, *Distributed Network Management with Dynamic Rule-Based Managing Agents.* Integrated Network Management V, Integrated Management in a Virtual World, Proceedings of the Fifth IFIP/IEEE International Symposium on Integrated Network Management, San Diego, CA, May 12—16, 1997. Lazar, A., Saracco, R., and Stadler, R. (eds.), London: Chapman & Hall, 1997, pp. 730—741.

www.pcltools.com

4.23 Further Reading

ITU-T Recommendation X.660 (ISO 9834-1), Information Technology, Open Systems Interconnection, Procedures for the Operation of OSI Registration Authorities: General Procedures, 1992.

ITU-T Recommendation X.660 (ISO 9834-1), Amendment 1, Information Technology, Open Systems Interconnection, Procedures for the Operation of OSI Registration Authorities: General Procedures, Amendment 1: Incorporation of Object Identifiers Components, 1996.

CHAPTER 5

Abstract Syntax and Transfer Syntax

5.1 Introduction

For two systems to communicate, each must understand the data sent from one to the other. This can be achieved by using a language that has the same syntax and semantics.

In the application layer, we use *abstract syntax,* which states only how data are arranged and what meaning they have. One of the possible abstract syntaxes is *Abstract Syntax Notation One* (ASN.1). Between the application layer and the presentation layer, a local set of rules can be used to transform data; however, the syntax of the data transferred between presentation entities must be understood by each end. This is known as *transfer syntax.* Abstract syntax and transfer syntax are negotiated at the beginning, during association time.

One transfer syntax is *basic encoding rules* (BER). BER state how data must be transformed before it is transferred to the other presentation entity. The local syntax can be purely dependent on the local protocols used. Figure 5-1 illustrates the concepts of abstract syntax and transfer syntax.

ITU-T Recommendations X.208 (Reference 5.1) and X.680 (Reference 5.2) describe standardized ways and steps to define data types and data values. Data types and data values are also referred to as *types* and *values,* respectively.

5.2 Abstract Syntax Notation One (ASN.1)

We will use a bottom-up approach to explain the concepts, starting with types and moving on to the definition of modules. In ASN.1, the starting

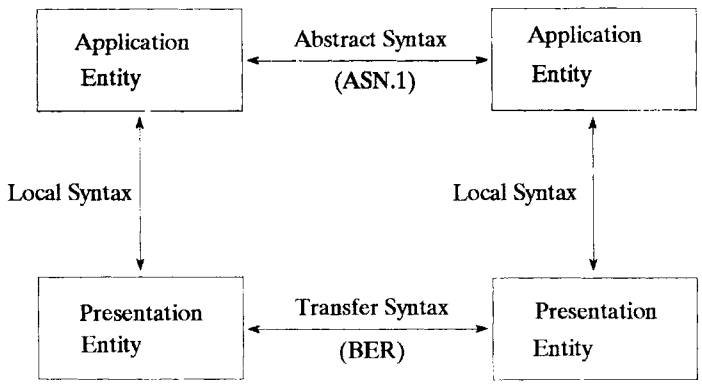


Figure 5-1
Abstract and transfer syntax concepts.

point is a *value*. An object instance can have a value. A collection of all these values is a *type*. These types are similar to the data types found in programming languages. Integer is an example of a data type. The whole idea in forming different data types is to define all possible simple data types first. Then these simple data types are combined in different manners to define complex data types.

X.208 is still widely used in TMN. Many commercially available ASN.1 compilers that perform syntax checking of ASN.1 definitions are yet to migrate to X.680. We will look into the topic of ASN.1 compilers in Chapter 12. ITU-T Recommendation X.680 is a refinement of X.208. It includes extensions and corrections to X.208. For this reason, we will discuss X.208 first. While examining X.208, we will not dwell much on the topics that have been dropped from X.680. Next we will state the differences between X.208 and X.680. This logical approach facilitates easier transition to the understanding of ASN.1 definitions to the X.680.

We broadly classify the ASN.1 built-in types as follows:

- Simple types
- Structured types
- Tagged types
- Subtypes

Like every formal programming language, ASN.1 has its own rules. Let us look into some of the important rules. They are explained as follows:

- *Keywords* appear in uppercase letters. An example of a keyword is:

```
BOOLEAN
```

- A *type* and *module name* begin with an uppercase letter. A type consists of one or more letters, digits, and hyphens. Examples of types are:

```
BOOLEAN (which is an ASN.1 built-in type)
HouseNumber (described next)
```

- A new type can be formed using ASN.1 built-in types. This is done by *type assignment*. An example of a type assignment is:

```
HouseNumber ::= INTEGER
```

  In the preceding definition, HouseNumber is a new type, `::=` means *defined as*, and INTEGER is the ASN.1 built-in type. HouseNumber is also known as the *type reference*. We notice here that HouseNumber starts with the uppercase letter H.

- An *identifier*, like a type, consists of one or more letters, digits, and/or hyphens. However, there is one difference between a type and an

identifier: The first character of an identifier starts with a lowercase letter. An example of an identifier is:

```
newModule
```

- A type can have one or more values. We can assign a value to a type by *value assignment*. An example of a value assignment is:

```
houseNumber HouseNumber ::= 234
```

In the preceding example, houseNumber is a *value reference* (used for providing a value to a type), HouseNumber is the type for which the value is provided, `::=` stands for the assignment of a value, and 234 is the actual value. Note that the value reference houseNumber starts with a lowercase letter. The syntax for a value reference otherwise is the same as for a type reference.

- A comment starts with -- and ends with either another -- or a period. An example of a comment is:

```
-- this is a comment in ASN.1 --
```

ASN.1 built-in types are displayed in Table 5-1. Let us examine each one of them with examples.

5.2.1 Simple Types

We can choose the BOOLEAN type for cases that have two states. As an example, the simple type *Normal* can be defined as follows:

TABLE 5-1

ASN.1 Built-in Types (X.208).

AnyType*	NullType	SetType
BooleanType	OctetStringType	SetOfType
BitStringType	ObjectIdentifierType	TaggedType
CharacterStringType	RealType	UsefulType (GeneralizedTime)
ChoiceType	SelectionType	UsefulType (EXTERNAL)
EnumeratedType	SequenceType	UsefulType (ObjectDescriptor)
IntegerType	SequenceOfType	UsefulType (UTCTime)

*Removed in X.680.

```
Normal ::= BOOLEAN
-- the whole thing is a production --
```

In this example, Normal is a BOOLEAN type with two values, TRUE or FALSE. This case is an example of a *production.* A production has a name on the left, followed by ::= and then a collection of sequences, which simply means a list of names. If there is more than one name, the names are separated by a vertical bar (|), which stands for *or,* as in many programming languages. Here, note that Normal starts with an uppercase letter because it is a type reference.

If we are not satisfied with just two states for a type, we can use an enumerated type to model the type. For example, assume that a type, EthernetAdapterStatus, has possible values of normal, degraded, off line, and failed.

```
EthernetAdapterStatus ::= ENUMERATED {normal (0), degraded (1),
offline (2), failed (3)}
```

In this production, note that ENUMERATED is used for three or more states, and the first state (normal) has a value of 0. So the next state (degraded) has a value of 1.

We want to have a counter that keeps track of the number of collisions in Ethernet over a fixed period of time. Let us call this counter *EthernetNumberCollisions.* This can be defined in ASN.1 syntax as follows:

```
EthernetNumberCollisions ::= INTEGER
```

INTEGER can be used to model an integer variable. It can also be used to define cardinal variables. Here, *cardinal variable* refers to values as a set or sequence. Now let us assume that EthernetNumberCollisionsRange is a range of 0 (no collisions) to 1000 collisions. Then *range* can be defined as follows:

```
EthernetNumberCollisionsRange ::= INTEGER {minimum(0),
maximum(1000)}
```

Both BIT STRING and OCTET STRING can be used to define binary data. The main difference between them is that in BIT STRING, the number of bits used is not necessarily a multiple of 8 bits, whereas in OCTET STRING, the length of the bits is in multiples of 8.

Let us go back to our example of EthernetAdapterStatus, which we used in explaining the ENUMERATED data type. Using BIT STRING, this example can be defined as follows:

```
EthernetAdapterStatus ::= BIT STRING {normal (0), degraded (1),
offline (2), failed (3)}
```

In the preceding example, EthernetAdapterStatus will have a value of 1000 B or 8 H (hex string) for the normal.

Let us investigate the CharacterStringType. CharacterStringTypes are:

- NumericString
- PrintableString
- TeletexString
- VideotexString
- VisibleString
- IA5String
- GraphicString
- GeneralString

Of these different CharacterStringTypes, NumericString, PrintableString, and IA5String are important. NumericString consists of digits 0 through 9 and spaces. PrintableString consists of letters, which can be uppercase or lowercase, digits, punctuation marks (",", ".", ":", etc.), and spaces. IA5String stands for International Alphabet Number 5 and is the same as the ASCII character set. For details on TeletexString, refer to Reference 5.3, for more on VideotexString, refer to Reference 5.4. The following are some examples of character strings.

```
AdapterCardType ::= PrintableString
  -- Here card type can be Ethernet, token ring, token bus.
AdapterCardType ::= NumericString
  -- Card type can be mapped as Ethernet to 0, token ring to 1, and
  so on.
```

Note that OCTET STRING may be used when an appropriate CharacterStringType is not available. As an example, we can define EthernetAdapterNumber as follows:

```
EthernetAdapterNumber ::= OCTET STRING
```

The NULL type is, by and large, used as a placeholder and is used if there is no element in a sequence.

The REAL type is used to model real numbers.

```
AdapterCardPrice ::= REAL
```

Because the adapter card price can be a specific number, for example $254.90, it is realistic to model it as REAL.

5.2.2 Structured Types

Let us assume that, in some cases, an Ethernet adapter does not have a number to identify it. In that case, it can be modeled as follows:

```
EthernetAdapterNumber ::= CHOICE {NULL, OCTET STRING}
```

Here, the use of CHOICE states that the Ethernet card number need not be there or an OCTET STRING. We will come back to the use of CHOICE later.

The SEQUENCE type is used to model a variable that has zero or more elements, in which the order of elements is important. The SEQUENCE type can have different types of elements, while the SEQUENCE OF type has only one type of elements.

```
EthernetCollisionsCounter ::= SEQUENCE
                                    {highValue INTEGER,
                                    lowValue INTEGER}
```

In this case, values in a collisions counter are expressed between the high and low values. Also, highValue and lowValue are just identifiers used for understandability; hence, lowercase letters are used as the initial characters. EthernetCollisionsCounter can also be expressed with the SEQUENCE OF type, because both the high and low values are mentioned with the same INTEGER type.

```
EthernetCollisionsCounter ::= SEQUENCE OF
                                    {highValue INTEGER,
                                    lowValue INTEGER}
```

Let us define one more counter for tokens lost in a token ring network.

```
TokenRingTokensLost ::= SEQUENCE OF
                            {highValue INTEGER,
                            lowValue INTEGER}
```

These two counters, EthernetCollisionsCounter and TokenRing TokensLost, can be combined into a LAN counter. Because the types are different, they are combined into a SEQUENCE type variable.

```
LanSimpleCounterLimits ::= SEQUENCE
       {ethernetCounter1 COMPONENTS OF EthernetCollisionsCounter,
tokenRingCounter1 COMPONENTS OF TokenRingTokensLost}
```

In the preceding definitions, COMPONENTS OF used with EthernetCollisionsCounter and TokenRingTokensLost indicates that all elements of both sequences are included.

The use of SET and SET OF types is similar to that of SEQUENCE and SEQUENCE OF, except in one respect. In SEQUENCE and SEQUENCE OF, we noticed that the order of elements is important. If the order of elements is not important, then SET and SET OF can be used in place of SEQUENCE and SEQUENCE OF, respectively. While using SEQUENCE, SEQUENCE OF, SET, and SET OF, context-specific tagging helps to identify each variable. Context-specific tagging is explained under Section 5.2.3.

SET OF can also be used for variables in which the elements are of the same type and the order of data is not important.

```
LanWorkstationSerialNumbers ::= OCTET STRING (SIZE (32))
LanSegment ::= SET OF LanWorkstationSerialNumbers
```

In this example, LanSegment is modeled as a collection of workstations, and each workstation is identified by its serial number. The serial number has a length of 32 octets (8 bits each).

Next, let us examine the use of SET. A LAN network may have combinations of Ethernet and token ring networks. The types used to model Ethernet and token ring networks are deliberately kept different in the following example. We assume that the order of these is not important.

```
MacAddresses ::= OCTET STRING (SIZE (6))
EthernetNetworks ::= SET OF MacAddresses
TokenRingNetworks ::= SET OF LanSegment
LanNetwork ::= SET
               {etherNet [0] IMPLICIT EthernetNetworks,
                tokenNet [1] IMPLICIT TokenRingNetworks}
```

In this example, MacAddresses has a length of six octets. The LanNetwork set has two elements: etherNet and tokenNet. In turn, etherNet is a collection of workstations identified by MacAddresses. However, in the case of tokenNet, there are sets of LAN segments, each of which in turn is a set of serial numbers as elements. IMPLICIT will be explained in the next section.

When only one of the alternative members of a collection has to be selected, then CHOICE is used. IMPLICIT tagging can be used if members belong to only a single data type.

```
ObjectName ::= CHOICE
               {localUniqueName GraphicString,
                localUniqueIdentifier     NumericString OPTIONAL}
```

This states that ObjectName can be in one of the following forms: localUniqueName or localUniqueIdentifier. Here, the keyword OPTIONAL states that localUniqueIdentifier may or may not be sent during data transfer to the other end, depending on circumstances.

Selection type makes use of the alternative types defined in CHOICE.

```
ObjectNameUsed ::= SEQUENCE
            {easilyReadableName localUniqueName <
             ObjectName}
```

In this example, we have used one of the alternatives: localUniqueName, modeled by the ObjectName type.

5.2.3 Tagged Types

A tagged type is used to model variables for removing ambiguities. In structured data types, which we have already seen, there are possibilities for confusion over such issues as how the receiving end interprets data, when it receives data on ObjectName, and whether the data refers to localUniqueName or localUniqueIdentifier. Here, we need tagging to explicitly state that the data sent from the other end refers to localUniqueName or localUniqueIdentifier, removing the confusion.

A tag can be EXPLICIT or IMPLICIT. By using an IMPLICIT tag, there is no need to transfer the data type during data transfer to the other end, while in the case of an EXPLICIT tag, transfer of the data type is required. This is understandable. For example, a CharacterString that identifies a variable can mean a Numeric String or a Graphic String. In order for a CharacterString to be clearly understood on the receiving end of the application entity, it is necessary to know exactly what is meant by the CharacterString.

If no IMPLICIT tag is specified, then it is assumed that an EXPLICIT tag is used. There is no mention of EXPLICIT in such cases. However, in module definitions, things are slightly different. If an IMPLICIT tag is used along with the module definition, then all the tags in the module are IMPLICIT. If a tag is left out or states that it is EXPLICIT, then it is assumed that the tagging used is EXPLICIT. Details on module definitions are provided in Section 5.2.4.

A user-defined tag has a class and a number within the square brackets []. The four user-defined tag classes are listed in Table 5-2. Let us examine each of them.

TABLE 5-2

ASN.1 User-Defined Tag Classes.

UNIVERSAL
APPLICATION
PRIVATE
CONTEXT-SPECIFIC

UNIVERSAL TAG. The UNIVERSAL tag is used for data types as provided in *X.208* (ISO 8824) (see Reference 5.1). The data types must be globally known and unique. Table 5-3 lists the different types of UNIVERSAL class tags. Let us look at an example of the UNIVERSAL class.

TABLE 5-3

Different UNIVERSAL Class Tags.

Tag	Type
UNIVERSAL 0	Reserved for use by encoding rules
UNIVERSAL 1	BooleanType
UNIVERSAL 2	IntegerType
UNIVERSAL 3	BitStringType
UNIVERSAL 4	OctetStringType
UNIVERSAL 5	NullType
UNIVERSAL 6	ObjectIdentifierType
UNIVERSAL 7	ObjectDescriptorType
UNIVERSAL 8	ExternalType and InstanceOfType*
UNIVERSAL 9	RealType
UNIVERSAL 10	EnumeratedType
UNIVERSAL 11	EmbeddedPDVType*
UNIVERSAL 12—15	Reserved for future editions of X.680
UNIVERSAL 16	SequenceType and SequenceOfType
UNIVERSAL 17	SetType and SetOfType
UNIVERSAL 18	NumericString
UNIVERSAL 19	PrintableString
UNIVERSAL 20	TeletexString (T61String)
UNIVERSAL 21	VideotexString
UNIVERSAL 22	IA5String
UNIVERSAL 23—24	Time
UNIVERSAL 25	GraphicString
UNIVERSAL 26	VisibleString (ISO646String)
UNIVERSAL 27	GeneralString
UNIVERSAL 28	UniversalString*
UNIVERSAL 30	BMPString*
UNIVERSAL 31	Reserved for addenda to X.680

*Added in X.680.

Assume that we are using an attribute, *Counter*, to keep track of beaconing in a token ring card. At a particular time, Counter may have a value of 21. This may be represented as an INTEGER. Here, INTEGER is the type.

```
Counter ::= [UNIVERSAL 2] IMPLICIT INTEGER
```

In this production, definition of the UNIVERSAL class tag is used. The INTEGER type is UNIVERSAL 2, as per Table 5-3.

APPLICATION TAG. The APPLICATION class is used to model the variables that are understood in the ASN.1 module being used. The presentation context understands the data type of the variable.

```
AnotherCounter ::= [APPLICATION 1] IMPLICIT INTEGER
```

In this example, [APPLICATION 1] is the tag. [APPLICATION 1] states that it is understood in the presentation context negotiated between the presentation entities, APPLICATION is the tag class and 1 is the number within the class. Here, IMPLICIT states that the explicit stating of AnotherCounter as an INTEGER is not necessary during data transfer. In this example, AnotherCounter is understood to be INTEGER by receiving the presentation layer entity.

CONTEXT-SPECIFIC TAG. In structured types, there is the problem of distinguishing between different elements. There can be major confusion, especially when using CHOICE. Unless elements are specifically mentioned, it is hard for the receiver to understand which element was sent. To remove these ambiguities, we use context-specific tags. The number for these tags starts with 0. For example:

```
BeaconingCounter ::= SET
                     {counterName  [0] IMPLICIT VisibleString,
                      counterNumber    [1] IMPLICIT INTEGER }
```

In this production, BeaconingCounter is defined as a SET. Mentioning that it is a SET means that order is not important. Here counterName has a context-specific tag of [0] and it states that it is the first element. Similarly, counterNumber has a context-specific tag of [1], and it is the second element.

PRIVATE TAG. The PRIVATE tag may be used to identify data types used within an organization or a country.

```
BecCounter ::= [PRIVATE 3] IMPLICIT INTEGER
```

In this example, [PRIVATE 3] has been used to specify that this is a widely used type within an organization or country.

OBJECT IDENTIFIER AND ObjectDescriptor. OBJECT IDENTIFIER is a set of names, numbers, or a mixture of the two associated with nodes from the root of the object identifier tree up to the object used, and it uniquely identifies an object. The object identifier tree is the same as the registration hierarchy tree (refer to Chapter 4, Figure 4.7). The root of the object identifier tree has three nodes: itu-t, iso, and joint-iso-itu-t. The basic principles of assigning identifiers are that an organization is responsible for the assignment of identifiers below it and the organization must ensure that the identifiers it issues are unique. As an example, iso, with a node value of 1, is responsible for assigning object identifiers below it. It has four children—standard, registration-authority, member-body, and identified-organization—with identifier values of 0, 1, 2, and 3, respectively. Identified-organization issues its own set of unique identifiers to the nodes below it. These parent and children nodes are connected by arcs.

Each value or name in OBJECT IDENTIFIER represents the values or names of nodes in the object identifier tree. Let us form an arbitrary OBJECT IDENTIFIER for lanNetwork:

```
lanNetwork OBJECT IDENTIFIER ::=
         {iso org dod internet private enterprises Xenterprises 85}
```

Here, org has a value of 3 for the node below the iso, dod has a value of 6 assigned to the node from org, internet has 1 for the node from dod, private has a label of 4 for the node from internet, and enterprises has 1 assigned to the node below private. Let us assume that Xenterprises has 140 given to its node and that in Xenterprises a value of 85 is assigned to lanNetwork. In this case, for the OBJECT IDENTIFIER of lanNetwork, instead of using the textual form as previously given, we can also use the numeric form as follows:

```
lanNetwork OBJECT IDENTIFIER ::= {1 3 6 1 4 1 140 85}
```

Please note that OBJECT IDENTIFIER can also have a mixture of names and values, instead of just names or just values as shown in these examples. The OBJECT IDENTIFIER of lanNetwork, as indicated here, is difficult to understand. So ObjectDescriptor, which represents easily readable texts, must be sent along with OBJECT IDENTIFIER. Sometimes

ObjectDescriptor may not end up being unique, but the combination of ObjectDescriptor and OBJECT IDENTIFIER will become unique globally.

```
etherlanNetwork ObjectDescriptor ::= [UNIVERSAL 7] IMPLICIT
GraphicString(SIZE(8))
```

Here, etherlanNetwork is the name of the ObjectDescriptor used and is a GraphicString of eight octets.

EXTERNAL. The EXTERNAL type is useful for modeling variables by abstract syntax other than ASN.1. This can refer to the types defined outside a particular module, and there are no restrictions on the use of the type. As can be seen from Table 5-3, the tag for EXTERNAL is UNIVERSAL 8.

```
DataTransferMode ::= EXTERNAL
```

Here, we are assuming that how we transfer the data is external to the module, which has DataTransferMode defined in it. We are assuming that for representation of data, we are using EBCDIC.

DATE AND TIME. Nations use different ways of reporting date and time. Thus, communication between different users around the globe is facilitated if the representation of date and time is standardized. The data type used for this purpose is *Time*, and the tags used are UNIVERSAL 23 for UTC (Coordinated Universal Time) and UNIVERSAL 24 for generalized time.

Three forms of generalized time are:

- *Calendar date:* 19940623235343.7. In this example, the first four digits represent the year (1994), the next two digits represent the month (06), the next two digits represent the day (23), the next two digits represent the hour (23), the next two digits indicate minutes (53), and the last digits indicate seconds (43.7). So, the date and time are June 23, 1994, at time 11:53:43.7 P.M. This form represents local time.
- *Time of day:* 19940623235343.7Z. All digits represent the same terms as they do in the local time, but the addition of the letter Z at the end indicates that the time refers to UTC time.
- *Time differential:* 19940623235343.7+1100. Here, local time is furnished as in the local time example. We can derive the UTC time from the time differential form of date and time representation. The UTC required is obtained by adding 11 hours to the local time.

Now let us examine universal time represented by UTC time. In this form, the date is represented by YYMMDD. Here, YY stands for the last two digits of a year, MM stands for the month, and DD represents the day of the month. The time can be of the form hhmm with four digits, or hhmmss with six digits. In this case, hh stands for hours (00 to 23), mm stands for minutes (00 to 59), and ss indicates seconds (00 to 59). This can be followed by either Z or a time differential that indicates the hours that must be added or subtracted to get the UTC time.

For example, 940623235343Z is similar to the UTC time listed under generalized time. One difference is that 1994 becomes 94, and seconds are not carried to decimal places (43 seconds instead of 43.7 seconds). The other form, 940623235343Z+1100, means that to get the UTC time we must add 11 hours.

5.2.4 Module Definitions

Module definitions are primarily used for grouping ASN.1 definitions. They also help in using type definitions defined in other places by making use of IMPORT and EXPORT mechanisms. Modules are analogous to functions in C language or subroutines in Pascal. There are module definitions in the definitions of managed object classes in standards and other documents. The following example illustrates this.

```
LanNetworkModule {iso org dod internet private enterprises Xenter-
prises 95}
   -- Above, LanNetworkModule is the module name, and {iso org dod
   -- internet private enterprises Xenterprises 95} is the assigned
   -- identifier.
DEFINITIONS EXPLICIT TAGS ::= BEGIN
   -- We have module body below.
IMPORTS
   RelativeDistinguishedName  FROM    InformationFramework
   {joint-iso-ccitt ds(5) modules(1)
    informationFramework(1)}
-- End of IMPORTS.
EXPORTS
   LanNetworkName ::= SEQUENCE of RelativeDistinguishedName
-- End of EXPORTS.
MacAddresses ::= OCTET STRING (SIZE (6))
LanWorkstationSerialNumbers ::= OCTET STRING (SIZE (32))
LanSegment ::= SET OF LanWorkstationSerialNumbers
EthernetNetworks ::= SET OF MacAddresses
TokenRingNetworks ::= SET OF LanSegment
LanNetwork ::= SET
   {etherNet   [0] IMPLICIT EthernetNetworks,
    tokenNet   [1] IMPLICIT TokenRingNetworks}
END
```

In the preceding definitions, IMPORTS states that RelativeDistinguishedName is defined in the InformationFramework and is used in this module. In the preceding module, LanNetworkName will be used in other module definitions. For this reason, it has to be mentioned in EXPORTS. Although it is not mandatory to have IMPORTS and EXPORTS, their use makes the definitions easier.

Notice that EXPLICIT TAGS immediately follows DEFINITIONS. We have deliberately introduced this to explain the idea behind using it. EXPLICIT TAGS is not necessary, because the default tag is "empty," which also means EXPLICIT TAGS. If we use IMPLICIT TAGS instead of EXPLICIT TAGS, then all of the tags we use in the module are IMPLICIT.

5.2.5 Subtypes

Subtypes, as the name suggests, make use of the existing types (see Figure 5-2). When subtypes are derived, they must have values. Subtypes are used in the following cases.

- When two *or* more types have common characteristics, subtyping makes the definition easier, if we include the common characteristics in a parent type. Subtypes include parent type and individual characteristics.
- If we want to limit the sizes or values of existing types, subtyping is helpful, making definitions of variables clear.
- If we wish to explain a subset with values in more detail, then subtyping is useful.

There are six subtypes. We will borrow some of the examples from previous sections to show how subtyping works.

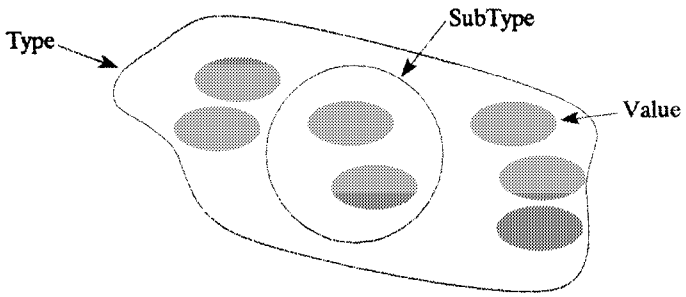


Figure 5-2
Types, values, and subtypes. (Source: Reprinted with permission from OSI Upper Layer Standards and Practices by Baha Hebrawi, copyright 1993 by McGraw-Hill.)

SINGLE VALUE. Single-value subtyping permits only one value out of many possible values of a subtype to be used.

```
TestResult ::= INTEGER (0 | 1 | 2)
```

In this case, we assign the values: pass is 0, fail is 1, and withdraw is 2. Thus, the results can be only one of these three states. The vertical bar (|) stands for *or.*

PERMITTED ALPHABET. Assume that house numbers can be only numbers that range in size from 1 to 5 digits. In such a case, we can come up with the following permitted alphabet subtyping:

```
HouseNumber     ::=     IA5String             (FROM
("0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9") SIZE(1..5))
```

CONTAINED. The contained subtype is helpful in forming a new subtype from the existing subtypes. In the example that follows, HouseAddress is a new subtype formed from subtype HouseNumber.

```
HouseAddress ::= INCLUDES HouseNumber
```

VALUE RANGE. Value ranges are used for INTEGER, REAL, and types obtained by tagging. As an example, assume that employee serial numbers are derived from integer values from 1000 to 20,000. A value range can be defined as follows:

```
EmployeeSerialNumber ::= INTEGER (1000..20000)
```

SIZE CONSTRAINT. A Size constraint can be used for forming subtypes and includes the keyword SIZE. SIZE mentions the length of the subtype derived from a parent.

```
LanWorkstationSerialNumber ::= OCTET STRING (SIZE (32))
```

Here, too, LanWorkstationSerialNumber is a subtype of the parent type OCTET STRING and has a length of 32 octets. Let us modify this example and assume that the workstation serial numbers are OCTET STRING with varying size anywhere from 5 to 32 octets. The modification is as follows:

```
LanWorkstationSerialNumber ::= OCTET STRING (SIZE (5..32))
```

INNER SUBTYPING. In the following example, we have made use of some of the inner subtyping keywords, such as WITH COMPONENTS, OPTIONAL, PRESENT, and ABSENT, to define subtypes of PureEtherLan and PureTokenLan. These two subtypes are derived from the parent, LanNetwork.

```
Bandwidth ::= INTEGER (1..4096)
-- Bandwidth can be from 1 to 4096 megabits per second
MacAddresses ::= OCTET STRING (SIZE (6))
LanSegment ::= SET OF LanWorkstationSerialNumbers
EthernetNetworks ::= SET OF MacAddresses
TokenRingNetworks ::= SET OF LanSegment
FDDIBackbonenetworks ::= SET OF MacAddresses
LanNetwork ::= SET
               {networkID            GraphicString,
               networkBandwidth      Bandwidth,
               fDDIBackNet           FDDIBackbonenetworks,
               etherNet              EthernetNetworks OPTIONAL,
               tokenNet              TokenRingNetworks OPTIONAL}
PureEtherLan ::= LanNetwork (WITH COMPONENTS
               {networkID            GraphicString,
               networkBandwidth      Bandwidth,
               fDDIBackNet           FDDIBackbonenetworks,
               etherNet              PRESENT tokenNet ABSENT})
PureTokenLan ::= LanNetwork (WITH COMPONENTS
               {networkID            GraphicString,
               networkBandwidth      Bandwidth,
               fDDIBackNet           FDDIBackbonenetworks,
               etherNet              ABSENT tokenNet PRESENT})
```


5.3 X.680

ASN.1-88/90 refers to the ASN.1 notation specified in CCITT Recommendation X.208 (1988) and ISO/IEC 8824:1990. Note that a module has to use one of the notations, and the module specifications have to clearly specify whether ASN.1-88/90 ASN.1 notation or current ASN.1 notation is used. Current ASN.1 notation refers to X.680 ASN.1 specifications.

5.3.1 X.208 versus X.680

There are some differences between X.208 and X.680 ASN.1 definitions. Some of the differences are described in the following text.

ANY REMOVED. In X.208, ANY is used when specifications are defined in another place before any data transfer is done. It is mainly a placeholder

for providing more details at a later time. The syntax of ANY is as given here:

```
AnyType ::= ANY | ANY DEFINED BY identifier
AnyValue ::= Type Value
This was changed for easier parsing of ASN.1 to
AnyValue ::= Type : Value
```

AnyType has been removed from the X.680 specifications. So, for placeholders, information object classes defined in X.681 (Reference 5.5) can be used.

MACRO REMOVED. In X.208, MACRO notation is used for locally defining the types and values of variables. It is also used for extending the ASN.1 grammar using available ASN.1 definitions. Instead of MACROs, this capability is provided by an information object class defined in X.681. However, the generality that was available due to MACROs is eliminated by the removal of MACROs.

Also, MACROs are used to define expressions and this capability is provided by the parameterization enhancement provided in X.683 (Reference 5.7).

5.3.1.1 Changes to Types and Values. In ASN.1 definitions in X.208, there are cases where there can be ambiguous definitions. So some of the definitions have been modified. Some of the changes are in the definitions of NamedType, ChoiceValue, and RealType. Let us look into each one of these.

NamedType. The X.208 ASN.1 definition of NamedType was

```
NamedType ::= identifier Type | Type | Selection Type
NamedValue ::= identifier Value | Value
```

In X.680, the *identifier* has been made mandatory for the NamedType and NameValue definitions. As a result, the syntax of NamedType has been changed as given below:

```
NamedType ::= identifier Type
NamedValue ::= identifier Value.
```

ChoiceValue. In X.208, the ASN.1 definition of ChoiceValue was

```
ChoiceValue ::= NamedValue
NamedValue ::= identifier Value | Value
```

In X.680, the ChoiceValue has been changed to

```
ChoiceValue ::= identifier ":" Value.
```

RealType. In X.208, RealType was defined as:

```
RealType ::= REAL
RealValue ::= NumericRealValue | SpecialRealValue
NumericRealValue ::= {Mantissa, Base, Exponent} | 0
Mantissa ::= SignedNumber
Base ::= 2 | 10
Exponent ::= SignedNumber
SpecialRealValue ::= PLUS-INFINITY | MINUS-INFINITY
```

There has been a slight change in the definition of NumericRealValue in X.680 and it is defined as

```
NumericRealValue ::= SequenceValue | 0
```

SequenceValue replaces the {Mantissa, Base, Exponent}. As a result of this change, now RealValue definition can be

```
SEQUENCE {
     mantissa INTEGER,
     base INTEGER (2|10),
     exponent INTEGER
     -- real number is mantissa multiplied by base raised to the
     -- power of exponent.
}
```

ADDITIONS. In X.680, some new data types have been added over the definitions provided in X.208. Some minor additions have also been made. The additions are explained in the following text.

AUTOMATIC TAGS. In X.208, TagDefault is defined as

```
TagDefault ::- EXPLICIT TAGS | IMPLICIT TAGS | empty
```

In X.680, AUTOMATIC TAGS has been added and this is applied at the module level. Here also, if TagDefault is empty then EXPLICIT TAGS is used. The modified definition of TagDefault is

```
TagDefault ::= EXPLICIT TAGS | IMPLICIT TAGS | AUTOMATIC TAGS | empty
```

UniversalString. UniversalString, which was not available in X.208, is a new addition in X.680. In Table 5-3, UNIVERSAL 28 has been given the universal class number for UniversalString. UniversalString includes all char-

acters available in ISO/IEC 10646-1. ISO/IEC 10646-1 has a varied range of characters such as control characters, graphic characters, and so on. For more details on the characters supported, refer to Reference 5.8. Let us take an example of UniversalString. To represent the Σ in Σ*stockvalue*, we can write

```
IMPORTS BasicLatin, greekCapitalSigma FROM ASN1-CHARACTER-MODULE
                                          {joint-iso-ccitt asn1(1)
specification(0) modules(0) iso10646(0)};
TotalSumOfStock ::= UniversalString {FROM (BasicLatin | greekCapi-
                                          talLetterSigma)}
myTotalSumOfStock TotalSumOfStock ::= { greekCapitalLetterSigma,
                                        "stockvalue"}
```

BMPString. BMPString is another addition in the X.680. Universal class number of 30 has been assigned for BMPString. BMPString is a restrictive subtype of UniversalString. BMPString uses the first 64K-2 cells of the characters in ISO/IEC 10646-1. An example of a BMPString is

```
tilde BMPString ::= (0, 0, 0, 126)
```

NEW BUILT-IN TYPES. New built-in types have been added in X.680. They are EmbeddedPDVType, ObjectClassFieldType, and InstanceOfType. EmbeddedPDVType is defined in X.680; the syntax of ObjectClassFieldType and InstanceOfType is explained in X.681.

EmbeddedPDVType has a universal class number of 11: PDV stands for presentation data value. Embedded PDV type is to be used for EXTERNAL type. This type is more suitable to model presentation layer type and data. EmbeddedPDVType type is more general in usage than EXTERNAL and there is no restriction that only ASN.1 has to be used. The data value of this type may or may not be the value of an ASN.1 type such as a value representing some item such as a graphic picture. This data type may include identification of the encoding rules used to encode the value used. The syntax of EmbeddedPDVType and values is

```
EmbeddedPDVType ::= EMBEDDED PDV
EmbeddedPDVValue ::= SequenceValue
```

An example of an EmbeddedPDVType is

```
WindowsFolder ::= SEQUENCE OF EMBEDDED PDV
```

ObjectClassFieldType AND InformationObjectClass. ObjectClassFieldType and InstanceOfType are explained in X.681. The ObjectClassFieldType is

used to define the types for information object class. Information object class contains a set of fields and these fields represent a collection of instances of the class. The syntax of ObjectClassFieldType is

```
ObjectClassFieldType ::= DefinedObjectClass"."FieldName
FieldName ::= PrimitiveFieldName"." +
```

The FieldName denotes either a type field or a variable type value field and is an open type notation. Here "+" stands for a production with a PrimitiveFieldName or an alternating series of production sequences starting and ending with PrimitiveFieldName. So, the sequence can be PrimitiveFieldName or PrimitiveFieldName.PrimitiveFieldName and so on.

```
ObjectClassFieldValue ::= OpenTypeFieldVal | FixedTypeFieldVal
OpenTypeFieldVal ::= Type ":" Value
FixedTypeFieldVal ::= Value
```

There are instances of object classes that have the values defined later or elsewhere. ANY and ANY DEFINED BY is used to define these instances in X.208. These have been replaced by information object classes in X.680 and X.681.

Let us define an information object class for ERRORS

```
ERRORS ::= CLASS
       {&OperationType        OPTIONAL,
        &Category             PrintableString (SIZE(8)),
        &ErrorCode            INTEGER UNIQUE}

WITH SYNTAX
       {[OPERATION-TYPE       &OperationType]
        [CATEGORY             &Category]
         ERROR-CODE           &ErrorCode}
```

In the preceding example, ERRORS is an object information class. ERRORS has &OperationType, a type field, and the two value fields &Category and &ErrorCode. In &OperationType the type field is undefined and is left open to be defined later by using OPTIONAL. In our example, ERRORS is the DefinedObjectClass and &OperationType is the FieldName. ERRORS.&OperationType is an example of ObjectClassFieldType.

WITH SYNTAX defines the syntax of the information object class, ERRORS. In the preceding definition, &ErrorCode is an identifier field as UNIQUE is mentioned here.

Let us define the possible categories of error codes as minor (1—5), major (6—10), critical (11—20), and failed (21—30). Each error code can be mapped to some meaningful textual description of errors. The range of

possible values is in parentheses. Let us define one information object operationError1 as given in the following example:

```
operationError1 ERRORS ::= {
                  {OPERATION-TYPE      INTEGER
                   CATEGORY            "Major"
                   ERROR-CODE           8}
```

After taking the example of one information object, let us define an OperationErrorSet information object set with four information objects.

```
OperationErrorSet ERRORS ::= {
                  {INTEGER "Major" 8} |
                  {REAL "Major" 10} |
                  {CHARACTER STRING "Minor" 1} |
                  {GeneralString "Critical" 15}}
```

OperationErrorSet can be represented in the form of a table; the values of parameters are shown in Table 5-4. An object class field type can be restricted to particular types or values by defining information object set. These restrictions are called *table constraints,* as there is an associated table as in the case of OperationErrorSet. Table constraints can be applied to ObjectClassFieldType or InstanceOfType. Different kinds of constraints that can be applied to information object classes are furnished in X.682 (Reference 5.6).

Let us look into how we can extract information from the OperationErrorSet.

```
errorCategory ERRORS.&Category ({OperationErrorsSet})
```

In the preceding example, errorCategory has the values in Table 5-4 of *Major, Minor,* and *Critical.* It can be noticed that *Failed* is not there in the OperationErrorSet. So by constraining, we have restricted the errorCategory to information objects that have errorCategory values of Major, Minor, and Critical.

TABLE 5-4

OperationErrorSet Table Value.

&OperationType	&Category	&ErrorCode
INTEGER	Major	8
REAL	Major	10
CHARACTER STRING	Minor	1
GeneralString	Critical	15

InstanceOfType. InstanceOfType has a universal class number of 8. InstanceOfType represents an instance of an information object class. The syntax of InstanceOfType is

```
InstanceOfType ::= INSTANCE OF DefinedObjectClass
```

An InstanceOfType needs an &id field, which is the OBJECT IDENTIFIER, and a value of &Type from an instance of the DefinedObjectClass. The InstanceOfType has an associated sequence type. The associated sequence type is used for defining the values and subtypes of InstanceOfType. The associated sequence type is defined as

```
SEQUENCE
      {
       type-id                <DefinedObjectClass>.&id,
       value                  [0]<DefinedObjectClass>.&Type}
```

The syntax of InstanceOfValue is

```
InstanceOfValue ::= Value
```

Let us take an example of InstanceOfType. Going back to the example of DefinedObjectClass ERRORS, the instance of an object class, ERRORS is defined as

```
INSTANCE OF ERRORS
```

The associated sequence type is now

```
SEQUENCE {
          type-id             ERRORS.&id,
          value               [0] ERRORS.&Type}
```

As we have mentioned earlier, InstanceOfType can also be constrained. The InstanceOfType can be constrained as follows:

```
INSTANCE OF ERRORS ({PossibleTypes})
```

And the associated sequence type now becomes

```
SEQUENCE {
           type-id  ERRORS.&id ({PossibleTypes}).
           value    [0] ERRORS.&Type ({PossibleTypes}) {@.type-id})}
```

PossibleTypes is a parameter, and this may not be resolved until implementation details are worked out. The type-id value is limited to one of the values permitted by PossibleTypes and value is the value of &Type

field of ERRORS. @ is used for AtNotation. AtNotation points to some other ASN.1 structure. As an example, "@" points to the ASN.1 defined within the associated SEQUENCE. So @.type-id indicates that the type-id referred means the type-id included in the innermost parent structure of SEQUENCE.

ASN.1 definitions furnished in X.681 (Reference 5.5), X.682 (Reference 5.6), and X.683 (Reference 5.7) primarily add flexibility to the definition of managed object classes and managed object instances.

PARAMETERIZATION. Parameterized definitions are used for filling holes at a later stage. There are many cases where the ASN.1 syntax cannot be fully defined. In such cases parameterized definitions can be used. ASN.1 parameterization rules are explained in X.683. Let us take a simple hypothetical example for a parameterized type. In this example, either a house name or a number can distinguish a house. For this example, the parameterized type is HOUSE-DISTINCTION {}.

```
HOUSE-DISTINCTION {ToBeDetermined} ::= CHOICE
            {houseName     ToBeDetermined,
             houseNumber   INTEGER}
```

In some other place, let us define the type as HOUSE-DISTINCTION {PrintableString}. In this case, the type notation becomes

```
CHOICE
            {houseName     PrintableString,
             houseNumber   INTEGER}
```

There are cases where extensions are required to ASN.1 definitions. Usually, software is developed in different releases. As an example, if we want to add some more features to the earlier release, we will have to change and/or add some more ASN.1 definitions to the existing ASN.1 definitions. For such cases, rules for extension of ASN.1 definitions have been added in Amendment 1 to X.680 (Reference 5.12). For more details about these rules, the reader should refer to Amendment 1.

5.4 Basic Encoding Rules (BER)

BER is used in the presentation layer, before the actual transfer of ASN.1 values, as shown in Figure 5-1. The data transfer is in the form of a stream of octets. The representation of data value as a sequence of octets is

known as *encoding*. When abstract syntax data is sent, encoding is done at the sending side and decoding is done at the receiving end.

X.209 defines the encoding rules. X.690 and X.691 have extended the encoding rules defined in X.209. Basic encoding rules present a set of options for encoding the data. Of the different available encoding options provided by BER, canonical encoding rules (CER) and distinguished encoding rules (DER) use one of the encoding options, eliminating ambiguity that may arise. DER is suitable for transmitting small encoded data values, and CER is suitable for transmitting a large amount of data or partial data. DER and CER are explained in X.690 (Reference 5.10).

The structure of BER can be of the form *identifier, length,* and *contents* (ILC) when the length of the contents is known. The identifier indicates the ASN.1 data type, the length supplies the length of the contents that follow the length field, and the contents contain the ASN.1 values to be transferred. This form is shown in Figure 5-3.

5.4.1 Identifier Field

The structure of the identifier field is shown in Figure 5-3. Class consists of the 8th and 7th bits and furnishes the tag used in data. Table 5-5 supplies the values of these bits for different classes. The 6th bit stands for either *primitive* or *constructed* (see Table 5-6), and bits 5 through 1 indicate

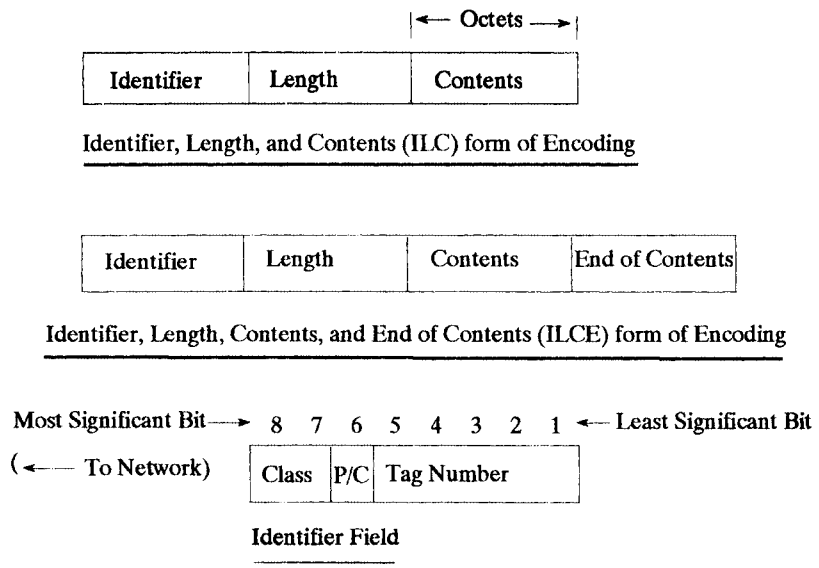


Figure 5-3
ILC and ILCE types of encoding.

TABLE 5-5

Class Bits in the Identifier Field.

Class	Bit 8	Bit 7
UNIVERSAL	0	0
APPLICATION	0	1
CONTEXT-SPECIFIC	1	0
PRIVATE	1	1

TABLE 5-6

P/C Values for Built-in ASN.1 Types.

Built-in Type	P, C, or P/C	Built-in Type	P, C, or P/C
BOOLEAN	P	CHOICE	P/C
INTEGER	P	Selection	P/C
BIT STRING	P/C	Tagged	P/C
OCTET STRING	P/C	ANY	P/C
NULL	P	EXTERNAL	P/C
SEQUENCE	C	OBJECT IDENTIFIER	P
SEQUENCE OF	C	Character String	P/C
SET	C	ENUMERATED	P
SET OF	C	REAL	P

P, primitive; C, constructed.

the ASN.1 tag number. In primitive encoding, content octets directly represent the value, whereas in constructed encoding, contents octets contain the encoding of one or more data values. In the P/C bit (Figure 5-3), a bit value of 0 indicates primitive encoding and a bit value of 1 represents constructed encoding. In Figure 5-3, bit 1 is the *least significant bit* and bit 8 is the *most significant bit*. When data is transferred, bit 8 is transferred first and bit 1 is transferred last. *Primitive* refers to simple atomic tags, while *constructed* data elements are formed from other data elements.

In tag numbers, bit 5 is the most significant bit and bit 1 is the least significant bit. Tags can be represented in two forms. If a tag number is 30 or less, then the short form with 5 bits is enough. In Figure 5-4, a tag number of 21 is shown.

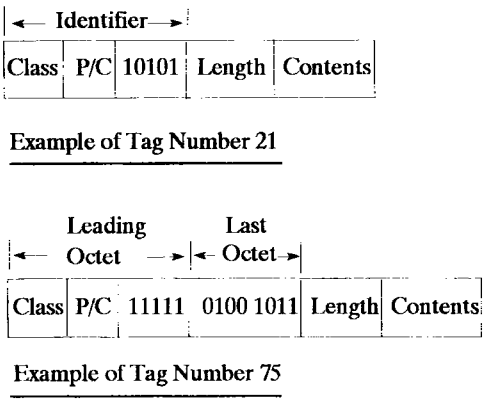


Figure 5-4
Example of tag numbers.

However, when the tag number is 31 or more, then 5 bits are not enough to represent them. In that case, the 5 bits of the leading octet are set to 1, and the tag numbers are represented by the unsigned integers in the subsequent octets. The 8th bit in each of the following octets except the last is set to 1. In the last octet, the 8th bit is set to 0 to indicate that this is the last octet to be used for calculating the tag number. In Figure 5-4, a high tag value (75) is shown.

5.4.2 Length Field

Length forms can be one of two types, *definite* or *indefinite*. The definite form, in turn, can be either *short* or *long*. This concept is depicted in Figure 5-5. In the length field, the 8th bit determines what form is used. If the

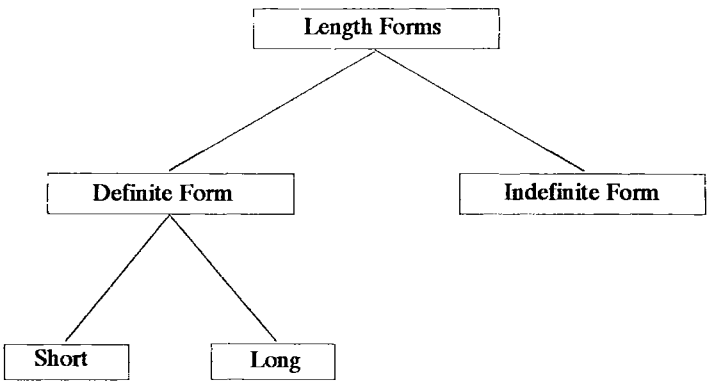
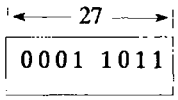
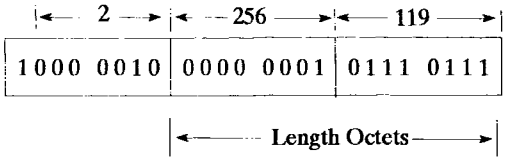


Figure 5-5
Different BER length forms.



Length Short Form - 27 Octets of Data Contents Field



Length Long Form - 375 Octets of Data Contents Field

Figure 5-6
Example of short-form and long-form lengths.

bit is 0, this indicates a short form. The 7 bits can hold up to a maximum of 127 octets. However, in the initial length octet, bits 7 through 1 cannot all be 1s, or "111 1111" B (127), because this is reserved for future extensions. Hence, only a maximum value of 126 can be used in the first octet that represents length. Figure 5-6 shows an example of 27 octets of data contents.

If more than 126 octets of length are to be indicated in the length field, then the long form is used. Here, the 8th bit is set to 1 in the first length octet. The subsequent bits from 7 to 1 furnish the number of length octets used.

As an example, the 375 length octets are shown in Figure 5-6. Here "000 0010" B indicates that there will be two additional octets that will have to be used for computing the length of the data octets. The bits in the subsequent two length octets represent 375.

If the length of the data contents is not known, then the *indefinite* form of encoding, also known as the ILCE form, is used. In the indefinite form, the 8th bit of the first octet of the length field is set to 1, and the rest of the bits from 7 to 1 are all set to 0. As usual, after the length field, we have data. After data, there are two octets of the end of contents field. The end of contents field is represented by "00 00" X, where X stands for a hexadecimal. We note here that because we have used "00 00" X for end of contents, we cannot have a value of "00 00" X in data.

5.4.3 Data Contents Field

Using examples, we will examine how built-in types are encoded.

BOOLEAN VALUE. A BOOLEAN value can be either TRUE or FALSE. The encoding of a BOOLEAN value of TRUE is shown in Figure 5-7. BOOLEAN belongs to the UNIVERSAL class, so bits 8 and 7 are both 0. Bit 6 is 0 because it is a primitive. The tag number for BOOLEAN from Table 5-3 is 1. Hence, the identifier field is represented as shown in Figure 5-7. The length of the data contents is one octet, so the length is shown as 1. The contents can be represented by "FF" X for the BOOLEAN value of TRUE. It may be noted that, instead of "FF" X, we can use any nonzero value in the first octet for a value of TRUE. For the BOOLEAN value of FALSE, the data contents octet is 0.

INTEGER VALUE. For the INTEGER value, the data contents field has twos complement. The INTEGER value is derived by including all bits from all octets. When computing the twos complement for a positive integer, the twos complement is the number itself. For a negative number, the ones complement is derived by reversing all the bits with 1 to 0 and bits with 0 to 1. Then the twos complement is obtained by adding 1 to the ones complement. Note that the operation is done on the binary representation of an INTEGER.

For the INTEGER value, too, we derive the identifier and length fields as described for the BOOLEAN value. For the data contents, 654 is represented as "028E" X and –654 is represented as "FD72" X, where X stands for hexadecimal representation.

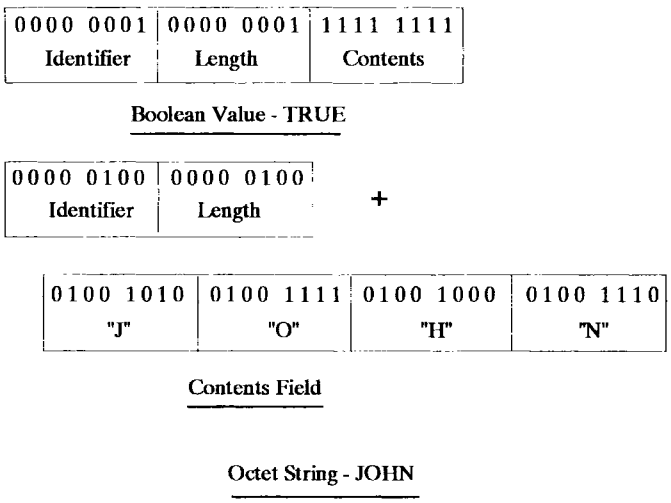


Figure 5-7 Encoding of BOOLEAN and OCTET STRING.

REAL VALUE. We use primitive encoding for REAL values. We use the tag nine of the UNIVERSAL class for encoding REAL values. If a REAL value is 0, then there is no contents field. A REAL value is given by:

```
REAL value=Mantissa(M)*Base(B)Exponent(E)
```

Encoding of an AdapterCardPrice of $254.90 can be done as {25490 10–2}. Here 25490 is the mantissa, 10 is the base, and –2 is the exponent. This is one of the six encoding schemes available for encoding REAL values [see X.208 (ISO 8824), Reference 5.1].

BIT STRING VALUE. As can be seen from Table 5-5, BIT STRING can be encoded either as a primitive or constructed. When transferring data contents, if it is necessary to transfer a part of the data, the constructed form can be used.

For primitive encoding, the BIT STRING identifier and length fields are derived in the same fashion as in BOOLEAN or INTEGER encoding. The data contents are broken as shown in Figure 5-8.

In Figure 5-8, in the data contents field, the first octet is known as the *initial octet.* There are other octets numbered from 1 to 5. The last octet is known as the *final octet.* The initial octet has the same number of unused bits as the final octet. This number, represented by an unsigned binary integer, can be from 0 to 7. BIT STRING occupies one to five octets.

Let us show how a BIT STRING value can be represented, using some examples. First, a BIT STRING of "0B2FADE" X is taken. In the primitive form, it is as follows:

```
Identifier      Length      Data contents
03              05          050B2FADE0
```

In this example, because BIT STRING belongs to the UNIVERSAL class, bits 8 and 7 are each 0, the sixth bit is 0, and the other bits are set to a tag number of 3. The tag number is 3 for BIT STRING, from Table 5-3. Hence, the identifier has a value of 3. The length of the data contents is 5. In the data contents, "05" X indicates that there are 5 unused bits in the final octet of "E0" X. The initial octet of the data content field has "05" X.

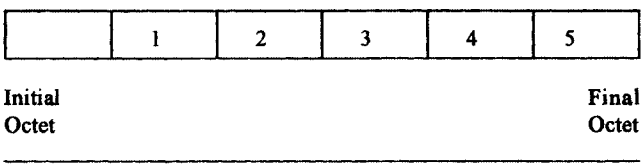


Figure 5-8
Data contents of BIT STRING values.

CONSTRUCTED FORMS. The same BIT STRING can be encoded in a constructed form as follows:

Identifier	Length	Data Contents	
23	0A		
	Identifier	Length	Data Contents
	03	03	000B2F
	03	03	05ADE0

This constructed string has two primitive BIT STRINGs. Here "00" (Data Contents column) in the first BIT STRING indicates that all bits in the final octet are used. However, "05" X (Data Contents column) in the second primitive BIT STRING shows that 5 bits are unused in the last octet of the data contents field. Identifier "23" X stands for "0010 0011" B. Here, bit 6 is 1, because it is a constructed string. The tag number is again 3, because it is a BIT STRING. Each primitive BIT STRING has a length of five octets. So the total length of the constructed BIT STRING comes to "0A" X.

INDEFINITE FORM. We have modified the same example to show how the BIT STRING "0B2FADE" X can be converted to BER using the indefinite form.

Identifier	Length	Data Contents	EOC
23	80	050B2FADE0	0000

In the length field, "80" X stands for the indefinite form, and it is represented as "1000 0000" B, followed by length, data in the form BIT STRING, and end of contents.

NULL VALUE. The NULL value is represented by a zero length octet, and the identifier field has "05" X.

OCTET STRING VALUE. OCTET STRING can be encoded as a primitive or constructor. The constructor form is used when we transfer data values in parts. Here all the octets are used for values, unlike the encoding of BIT STRING values. Using OCTET STRING, "JOHN" will be encoded as shown in Figure 5-7. Here, we use the primitive form, and the UNIVERSAL class number for OCTET STRING is 4.

CONSTRUCTED FORMS. As can be seen from Table 5-6, SEQUENCE, SEQUENCE OF, SET, and SET OF use constructed forms. Due to this, a P/C value of 1 is used for constructor. There are no firm rules on how different elements must be transmitted by a sender in the case of SET and SET OF. As we have seen earlier, in SET and SET OF definitions of ASN.1,

no restrictions are placed on the order of elements. On the contrary, for SEQUENCE and SEQUENCE OF, the order of elements is important. So, during formation of transfer syntax, the order is preserved. The order of elements in ASN.1 is also preserved, if the values are sent for OPTIONAL or DEFAULT for all the cases. In this case, there is no distinction between SET, SET OF, SEQUENCE, and SEQUENCE OF.

OBJECT IDENTIFIER AND OBJECTDESCRIPTOR. In the case of OBJECT IDENTIFIER, we encode it after we make a conversion of the first two elements into one by using the first element, × 40, plus the next element. Each element uses bits 7 to 1 of an octet. If an element cannot be represented by 7 bits of an octet, more octets are used. Bit 8 of the first octet and each subsequent octet except the last one is set to 1. Bit 8 of the last octet is set to 0, indicating that it is the last octet used for calculating the value of an element. Let us return to our example of lanNetwork and see how an OBJECT IDENTIFIER is encoded.

```
lanNetwork OBJECT IDENTIFIER ::= {1 3 6 1 4 1 140 85}
```

The lanNetwork is encoded as shown in Figure 5-9. Because OBJECT IDENTIFIER falls under the UNIVERSAL class, bits 8 and 7 are both set to 0. Because it is primitive, bit 6 is 0. OBJECT IDENTIFIER has a value of 6 in the UNIVERSAL class, and so the tag number is 5. The first element of OBJECT IDENTIFIER for encoding is 1 × 40 + 3 or 43, so for encoding we use {43 6 1 4 1 140 85}. Each element is encoded using one or more octets. In Figure 5-9, we have used two octets to encode the value of 140.

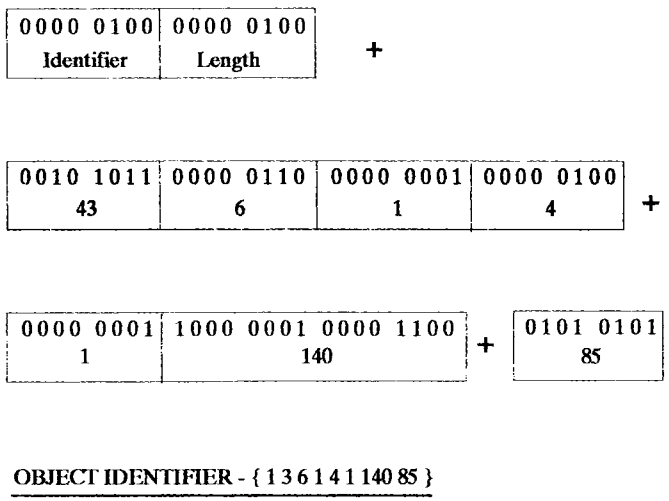


Figure 5-9
Example of encoding of OBJECT IDENTIFIER.

Note that, in the first octet, in order to represent 140, the 8th bit is set to 1 and in the second octet the 8th bit is set to 0.

ObjectDescriptor is encoded as an OCTET STRING and the UNIVERSAL class value of 7 is used.

EXTERNAL. EXTERNAL has relevance if it is defined, since data types other than ASN.1 may be used. There must be agreement on the encoding between the sending and receiving sides for the data to be useful.

CANONICAL ENCODING RULES (CER). CER places some restrictions on the BER. If the encoding is constructed, the indefinite length form (Figure 5-5) is used. However, if the encoding is primitive, then fewest length octets are used.

Bit string, octet string, and restricted character string use primitive encoding if there are 1000 or less octets in the contents field. However, when there are more than 1000 octets, constructed encoding is used.

For encoding set component values, canonical order of tagging is used. In a canonical order of tags, elements with universal class tags appear first, followed by application tags, context-specific tags, and finally private class tags. Additionally, within a class of tags, the elements appear in the ascending order of their tag numbers.

Component values of a set value are encoded in order of their tags; the order of encoding is from the lowest tag to the highest, and the encoding of tags follow the canonical order of tagging. An untagged choice is treated as having the lowest tag.

DISTINGUISHED ENCODING RULES (DER). DER also uses the encoding rules used in BER with some restrictions, just like CER. In DER, a definite length form of encoding is used. For bit string, octet string, and restricted character string types, only primitive encoding is used, and encoding of component values of a set value follows the canonical order of tagging.

In addition to these specific rules, CER and DER follow some more rules. If the BOOLEAN value is TRUE, then all bits of contents of a single octet are set to 1 as shown in Figure 5-7. When encoding set or sequence values, component values set to default values are not encoded. For more details on the encoding CER and DER, refer to Reference 5.10.

When we encode and decode the abstract syntax, we need to know which encoding rules we are using. To identify whether BER, CER, or DER encoding is used, unique object identifiers and object descriptors have been assigned. The object identifiers and object descriptors are:

- *BER:* {joint-iso-itu-t asn1(1) basic-encoding (1)} and "Basic Encoding of a single ASN.1 type"
- *CER:* {joint-iso-itu-t asn1(1) ber-derived (2) canonical-encoding (0)} and "Canonical encoding of a single ASN.1 type"
- *DER:* {joint-iso-itu-t asn1(1) ber-derived (2) distinguished-encoding (1)} and "Distinguished Encoding of a single ASN.1 type"

5.5 Notes on the Use of ASN.1 and BER

In some cases, the encoding of ASN.1 types is not efficient. As an example, to encode a BOOLEAN value, three octets are required. Such lacunas can be overcome by using *Packed Encoding Rules* (PER). The PER compact encoding scheme enables us to represent the values by eliminating the use of identifier, length, or both, depending on the situation. For more details on PER refer to X.691, Specification of Packed Encoding Rules (PER) (Reference 5.11).

The use of ASN.1 and BER is not mandatory in applications for transferring data between application entities by OSI standards. It may or may not be used, depending on the applications.

One disadvantage of ASN.1 and BER is that they are difficult to understand. There are encoders that convert ASN.1 values to BER. The encoded data can be converted back to ASN.1 values using decoders. These encoders and decoders are commercially available. Note that encoder and decoder terms have become common in the industry and they are used for conversions from one format to another, but in this book they specifically refer to the data conversions from ASN.1 to BER and from BER back to ASN.1. These encoders and decoders add complexity and extra processing and code. Extra processing has an effect on performance and should be avoided as much as possible.

Encoders and decoders simplify the coding aspect, but they do not eliminate the extra steps involved in encoding and decoding. So, why do we need these transformations if both the ends speak the same language? A partial solution is to use the transformation only where needed. There may be cases in which the application entities span different languages and continents. In such cases, encoding and decoding may be useful.

5.6 Summary

In this chapter, we have investigated ASN.1, used for communicating between application entities. We have discussed both X.208 and X.680. ASN.1 concepts have been explained with examples. We also looked at the BER, used for transferring data between presentation layers. After the discussion of BER (X.209), we have also examined CER and DER. Finally, we have discussed the practical aspects of using ASN.1 and BER.

5.7 References

5.1. ITU-T Recommendation X.208 (ISO/IEC 8824), Information Processing Systems, Open Systems Interconnection, Specification of Abstract Syntax Notation One (ASN.1), 1988.

5.2. ITU-T Recommendation X.680 (ISO/IEC 8824-1), Information Technology, Abstract Syntax Notation One (ASN.1), Specification of Basic Notation, 1994.

5.3. ITU-T Recommendation T.61, Character Repertoire and Coded Character Sets for the International Teletex Service, 1992.

5.4. ITU-T Recommendation T1.101, Data Syntax 1 for International Interactive Videotex Service, 1992.

5.5. ITU-T Recommendation X.681 (ISO/IEC 8824-2), Information Technology, Abstract Syntax Notation One (ASN.1), Information Object Specification, 1994.

5.6. ITU-T Recommendation X.682 (ISO/IEC 8824-3), Information Technology, Abstract Syntax Notation One (ASN.1), Constraint Specification, 1994.

5.7. ITU-T Recommendation X.683 (ISO/IEC 8824-4), Information Technology, Abstract Syntax Notation One (ASN.1), Parameterization of ASN.1 Specifications, 1994.

5.8. ISO/IEC 10646-1, Information Technology, Universal Multiple-Octet Coded Character Set (UCS): Architecture and Basic Multilingual Plane, 1993.

5.9. ITU-T Recommendation X.209 (ISO/IEC 8825), Open Systems Interconnection, Specification Basic Encoding for Abstract Syntax Notation (ASN.1), 1988.

5.10. ITU-T Recommendation X.690 (ISO/IEC 8825-1), Information Technology, ASN.1 Encoding Rules, Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER), 1994.

5.11. ITU-T Recommendation X.691 (ISO/IEC 8825-2), Information Technology, ASN.1 Encoding Rules, Specification of Packed Encoding Rules (PER), 1995.

5.12. ITU-T Recommendation Amendment 1 to X.680 (ISO/IEC 8824-1), Information Technology, Abstract Syntax Notation One (ASN.1), Specification of Basic Notation, Amendment 1: Rules of Extensibility, 1995.

5.8 Further Reading

Hebrawi, B., *OSI Upper Layer Standards and Practices,* New York: McGraw-Hill, 1993.

ISO 6093, Information Processing Systems, Representation of Numerical Values in Character Strings for Information Interchange Version, 1985.

Rose, M. T., *The Open Book, A Practical Perspective on OSI,* Englewood Cliffs, NJ: Prentice Hall, 1990.

CHAPTER 6

Structure of Management Information and TMN Information Model

6.1 Introduction

We have seen in earlier chapters that for a resource to be managed, it must be represented as a managed object class. When representing a new resource, the inheritance hierarchy defined by the X.721, Definition of Management Information (Reference 6.2), is very useful. The new resource must be appropriately made a subclass of the managed object class, which it closely resembles. All the characteristics, such as attributes, operations, notifications, and behaviors, of the superclasses are inherited. Sometimes an inheritance property reduces the process of defining a managed object class to a minor tweaking of characteristics.

It is also useful to understand how a managed object class is defined, which makes it essential that the concepts involved in the definitions of managed object classes be understood. We have investigated the meaning of a managed object class in Chapter 1. The present chapter will extend the concepts explained earlier by further examining how a managed object class is defined.

The most important activity in TMN is appropriate modeling of the telecommunications resources as managed object classes. From this point of view, we need to take a close look at how to define the managed object classes. Guidelines for the definition of managed objects (GDMO) are used for defining the managed object classes. We first look into the concepts used in defining the managed object classes. As M.3100, Generic Network Information Model (Reference 6.10), is specifically devoted to the generic TMN-related managed object classes, we will also look into the different managed object classes defined in M.3100.

6.2 Overview of the Structure of Management Information (SMI) Documents

ITU-T X.720, Management Information Model (Reference 6.1), primarily furnishes basic building blocks for the definition of a managed object class. The X.721 contains the following:

- Concepts of managed object classes and how they are defined
- Explanations of how managed object classes should be defined for purposes of compatibility and interoperability

- Descriptions of management operations that can be done on attributes and managed objects
- Descriptions of filter operations that can be done on managed object classes, because filters are widely used in CMIP
- Descriptions of notifications that must be emitted by managed objects
- Discussion of the issues involved in the naming of managed objects—including the concept of containment—which are very important for consistency in implementation

X.721, Definition of Management Information (Reference 6.2), defines the managed object classes required for systems management functions. The most important standard, X.722, Guidelines for the Definition of Managed Objects (Reference 7.3), specifies rules and provides templates for defining managed object classes. This document is popularly known as *Guidelines for the Definition of Managed Objects* (GDMO).

X.723, Generic Management Information (Reference 7.4), defines managed object classes that can be used in different layers. This document also provides definitions of resource-specific managed object classes such as ports. X.724, Requirements and Guidelines for Implementation Conformance Statement Proformas Associated with OSI Management, relates to the conformance testing of managed objects and systems management functions. To those involved in the TMN area, it is useful to know what is in each document; thus, we have furnished further details on these documents in Section 6.5.

6.3 Managed Object Class

The definition of a managed object class needs the following:

- An *identifier* of the managed object class
- An *allomorph attribute* that identifies the managed object classes that are allomorphic to the managed object class defining the allomorph attribute
- A *name binding attribute*, used to uniquely identify a managed object of the managed object class, that states the relationship of a managed object to its superior
- A *package attribute*, which is a conditional attribute and contains a list of object identifiers of the packages used

6.4 Guidelines for the Definition of Managed Objects (GDMO) Templates

To avoid confusion in defining managed object classes, a formal GDMO format for definitions must be used. This standard format is known as a *template*. These templates are used for the definitions of components of managed object classes such as packages, parameters, attributes, attribute groups, behaviors, actions, or notifications. GDMO provide the following templates:

- *Managed object class:* In this template, inheritance relationships, which are central to the reuse of characteristics with other managed object classes, are defined. Managed object classes contain packages of behavior, attributes, attribute groups, actions, and notifications. Also, additional templates can be included or borrowed from other managed object classes. Figure 6-1 details the contents used in the definition of a managed object class.
- *Package:* Attributes, attribute groups, operations, notifications, behavior definitions, and parameters are collected to form an identifiable template. A package template can be inserted in managed object class templates.

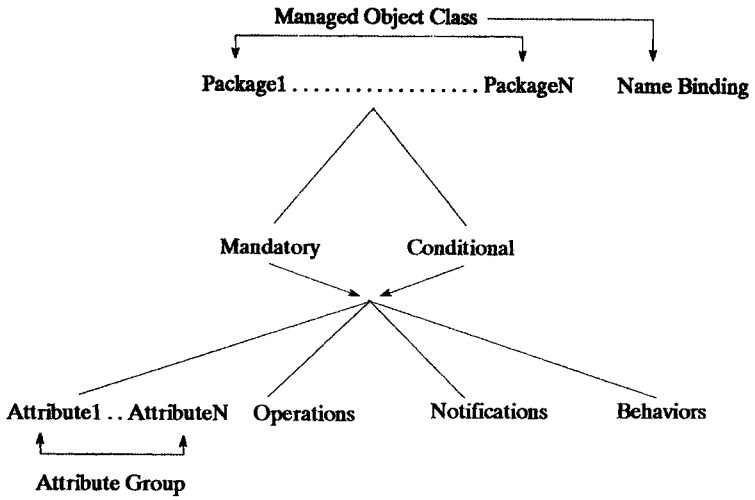


Figure 6-1
Managed object class definition.

- *Attribute:* This template is used for providing attribute syntax, and includes attribute syntax, rules to test the attribute values, behaviors, the attribute identifier, and parameters.
- *Attribute group:* When attributes are grouped for convenience, they form attribute groups. Attribute group templates indicate the set attributes comprising the group, and an identifier value to identify the attribute group.
- *Action:* This template is used to define the behavior and syntax of action types, which are carried in CMIS M-ACTION.
- *Behavior:* This template is used to extend the semantics of previously defined templates. It is helpful in further explaining managed object classes, name bindings, attributes, parameters and actions, and notifications that have been defined elsewhere.
- *Notification:* Notifications carried in CMIS M-EVENT-REPORT are defined in this template.
- *Parameter:* Parameters used in defining attributes, operations, and notifications are defined in this template. Specifications and parameter syntaxes are also listed along with the behaviors.
- *Name binding:* This template is used for uniquely naming a managed object. It specifies the naming attribute used for naming and identifies the superior object.

We have used a bottom-up approach in furnishing the details of these templates, starting with the explanation of attributes (Section 6.4.1) and proceeding to the managed object classes (Section 6.4.10). When using templates for defining managed object classes, certain conventions are followed; these are furnished in X.722. The important ones are:

- A semicolon (;) marks the end of each construct and the end of a template.
- All symbols and keywords are case-sensitive. That means lower- and uppercase letters indicate different things. For example, "A" is different from "a."
- Comments start with a double hyphen (--) and end with a double hyphen or the end of a line.
- Spaces, the end of a line, a blank line, or comments are valid delimiters.
- Whenever text is used in a template, one of the text-delimiter characters is used. These are: ! " # $ ^ & * ' ` , ? @ \. However, the same text delimiter should be used at the start and end of the string represent-

ing the text. For example, if we use an exclamation point (!), then the text should also end with an exclamation point.

- The template label must be unique to a document. When defining templates, the syntax is <template-label> Template Name.

Strings included within square brackets ([]) may be present or absent in each instance of a template; these strings delimit parts of the template definition. If the square brackets are followed by an asterisk (*), as in the case of []*, the contents within the square brackets may appear zero or more times.

6.4.1 Attribute

A managed object class must have properties to be meaningful. These properties are known as *attributes*. An attribute must have a value: For example, the definition of a managed object class, tokenRing, can have an attribute such as tokenRingBandwidth. When defining this attribute, tokenRingBandwidth=16 indicates a token ring with a bandwidth of 16 Mbits.

Attributes can be single-valued or set-valued. *Single-valued* attributes have only one value. *Set* is a mathematical concept. *Set-valued* means that there can be more than one value of the same type. In sets, ordering is not important, and there is no repetition of values.

An attribute may have more than one value. For example, assume that the managed object class tokenRing has an attribute tokenRingBandwidth. We model bandwidth as an attribute, because bandwidths can vary. In tokenRingBandwidth=4,16,100, the attribute tokenRingBandwidth has different values, namely 4, 16, and 100 Mbits.

Attribute values are visible at the boundary of a managed object class. When we perform certain operations on a managed object class, these values can be retrieved or modified. For example, a GET on the attribute tokenRingBandwidth will retrieve the values of 4, 16, or 100. Similarly, when we do a SET on the attribute tokenRing Bandwidth, we can modify a value from 16 to 4.

When an object is identified, the managed object class must have at least one attribute used for naming. This is a mandatory attribute. This attribute identifier and its value uniquely identify a managed object. The attribute is read-only. If this attribute can be deleted, then it is necessary to define an additional unique identification attribute.