



P R O D U C T O V E R V I E W

Mediator™ 3.0

Xenadyne, Inc.
2207 West 24th Street
Los Angeles, California, 90018 USA
Phone: 323 732-7248
Web: <http://www.xenadyne.com>
Email: info@xenadyne.com

© 2001 XenadyneInc.
© 1995-2000 ObjectStreamInc.
All rights reserved.

Xenadyne Inc
2207 West 24th Street
Los Angeles California 90018
Telephone: (323) 732-7248
Email: info@xenadyne.com
Web: <http://www.xenadyne.com>

MARCH 2004

ObjectStream, and ObjectStream Mediator are trademarks of ObjectStream, Inc.
All other products, services, or company names mentioned herein may be trademarks or
registered trademarks of their respective owners.

The contents of this document and the products it describes are subject to change without notice.

CONTENTS

Chapter 1 Mediation Concepts

What is Mediation?	1-1
Where Can Mediation Occur?	1-3
Xenadyne Mediator Implementation	1-4

Chapter 2 Xenadyne Mediator

Product Architecture	2-1
Integrating Xenadyne Mediator	2-6

Chapter 3 Mediator NEDL Information Model

Overview	3-1
Defining an NE's Resources and Attributes	3-2
Mapping Multiple NE Types	3-4
Specifying Operations and Access Policy	3-4
NEDL Files and the NEDL Language	3-4

Chapter 4 NEDL Development Environment

Building NEDL Files	4-1
Generating Reports	4-1
NEDL Library	4-4
Test Scripting Language	4-5
Simulators	4-9

Chapter 5 Mediator MOI

Mediator MOI C API	5-2
--------------------------	-----

Chapter 6 Technical Details and References

Technical Details	A-1
Technical References	A-3

Mediation Concepts

What is Mediation?

Mediation, by definition, transforms or “mediates” information passed between two incompatible entities (see Figure 1-1). The mediation component receives information from one entity, transforms the information as necessary, and forwards the information to the other entity in a form that the target entity can understand.

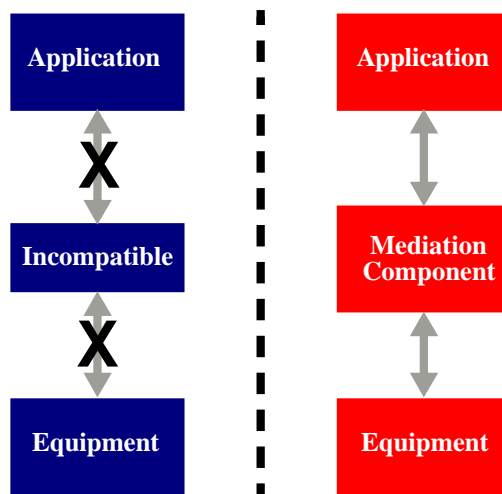


Figure 1-1. General Mediation

Mediation is required for network management environments because:

- Telecommunications and enterprise networks are composed of hundreds of different network elements manufactured by various vendors.
- Network Elements (NEs) and Enterprise Management System (EMS)/Network Management System (NMS)/Operations Support System (OSS) applications employ a range of non-compatible information models.
- NEs employ different communication protocols for supporting different kinds of services.

Note: Throughout this document, managed entities are referred to as NEs for simplicity and consistency. In reality, however, a managed entity could also be an EMS, OSS, etc. You should substitute the managed entity of your choice as needed.

Mediation Concepts

What is Mediation?

Mediation can be employed for "normalizing" data between information models (for example, GDMO, SNMP MIBs, and proprietary information models). Mediation can also be used to convert data between protocols (for example, TL1, ASCII, SNMP, and QD2). In most implementations, mediation is used to provide a combination of data normalization and protocol conversion for a heterogeneous network (see Figure 1-2).

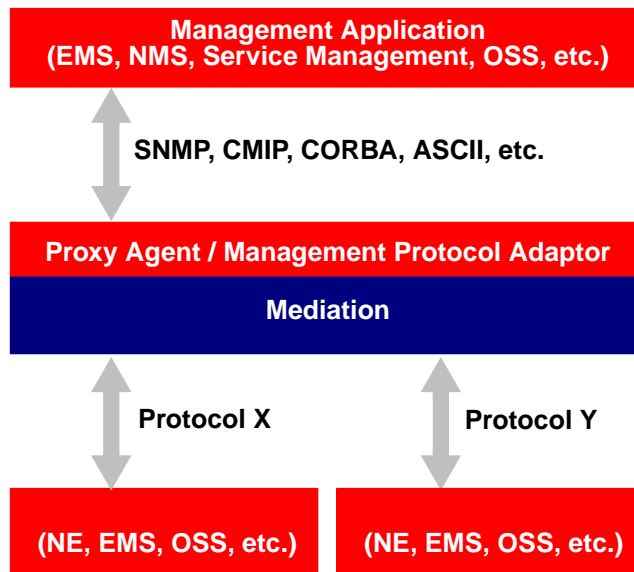


Figure 1-2. Multi-Protocol Mediation

Configurable Mediation

NEs and the systems that support them are dynamic by nature. Frequent updating of NEs is required to enable the introduction of new features and services. As NEs evolve, the mediation component must also be updated to adapt to these changes. Therefore, in order for a mediation component to be useful it must be able to support changes in the rules it uses for data normalization and protocol conversion. In a word, it must be *configurable*.

Dynamically Configurable Mediation

In addition to being configurable, mediation must also be *dynamic*. The alteration of the existing set of rules used by the mediation component to perform the data and protocol translation must be simple to implement and incremental in nature. The mediation component must also be able to handle the rapid introduction of an entirely new model or protocol without interrupting the mediation functions it is performing for other NEs or systems.

Many telecommunications providers currently have mediation as part of their management systems, which typically is "hard coded" into the management system. Because any change to the mediation layer typically requires that the system be brought down, re-coded, re-compiled and then re-initialized, these systems require extensive maintenance and engineering support to remain in service. Any change in any part of the management system—from adding new NEs to simply changing one of the messages—results in a code rewrite. The revision process is time-consuming and is costing many telecommunications providers the ability to quickly deploy equipment and systems.

Dynamically configurable mediation is the ability to modify the transformation rules while the mediation system is operational.

Where Can Mediation Occur?

Mediation can occur in a variety of scenarios, as illustrated in Figure 1-3.

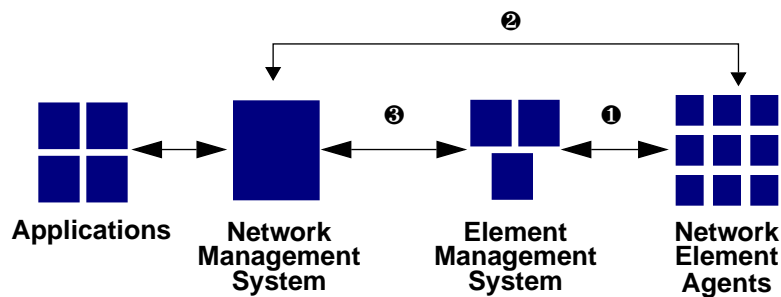


Figure 1-3. *Where Mediation Occurs*

Mediation can occur:

- ❶ Between NEs and the Element Management System (EMS)
- ❷ Between NEs and the Network Management System (NMS)
- ❸ Between the EMS and NMS

Additionally, mediation can be embedded directly within either entity (see Figure 1-4). For example, mediation can be embedded in either the NE or the EMS to perform mediation between the NE and EMS.

Xenadyne Mediator Implementation

Figure 1-4 and the following subsections describe how the product typically fits into a heterogeneous network management environment.

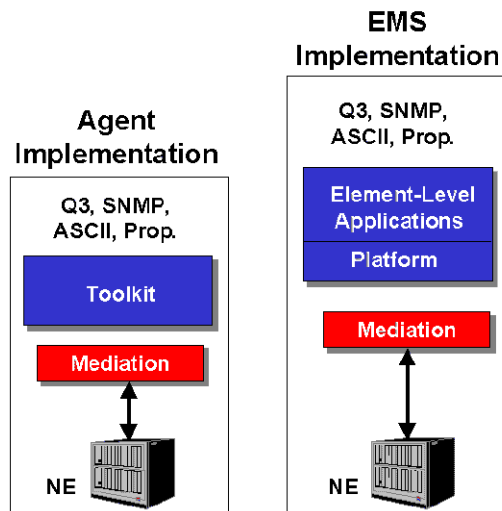


Figure 1-4. Mediation Implementation

Agent Implementation

Xenadyne Mediator can be integrated with a management agent toolkit. When integrated with a CMIP agent toolkit, for example, Xenadyne Mediator allows the monitoring and controlling of TL1, ASCII, binary, or SNMP NEs from a CMIP/Q3-based network management system. Similarly, Xenadyne Mediator can also be integrated with SNMP Agent toolkits, allowing an SNMP-based management system to monitor and control ASCII NEs.

EMS Implementation

Xenadyne Mediator can support fault, configuration, accounting, performance, and security management (FCAPS) applications when integrated with an EMS. Xenadyne Mediator isolates the changes that occur in the NE layer from upstream EMS processes. This isolation allows the network management layer to continually evolve to meet the business and service needs of the operator, without requiring the management system to undergo extensive alterations. Any changes in the specific software load of the NE are reflected in the Rosetta file.

Applications in a CORBA Environment

Xenadyne Mediator can form the core of a Common Object Request Broker Architecture (CORBA) Event Server. In integrating the Mediator in this manner, it can perform the necessary translation between NEs and the Event Server. The Mediator provides all the necessary connection management and mediation functions, which greatly simplifies the implementation of an Event Server for a heterogeneous network. The general architecture is illustrated in Figure 1-5.

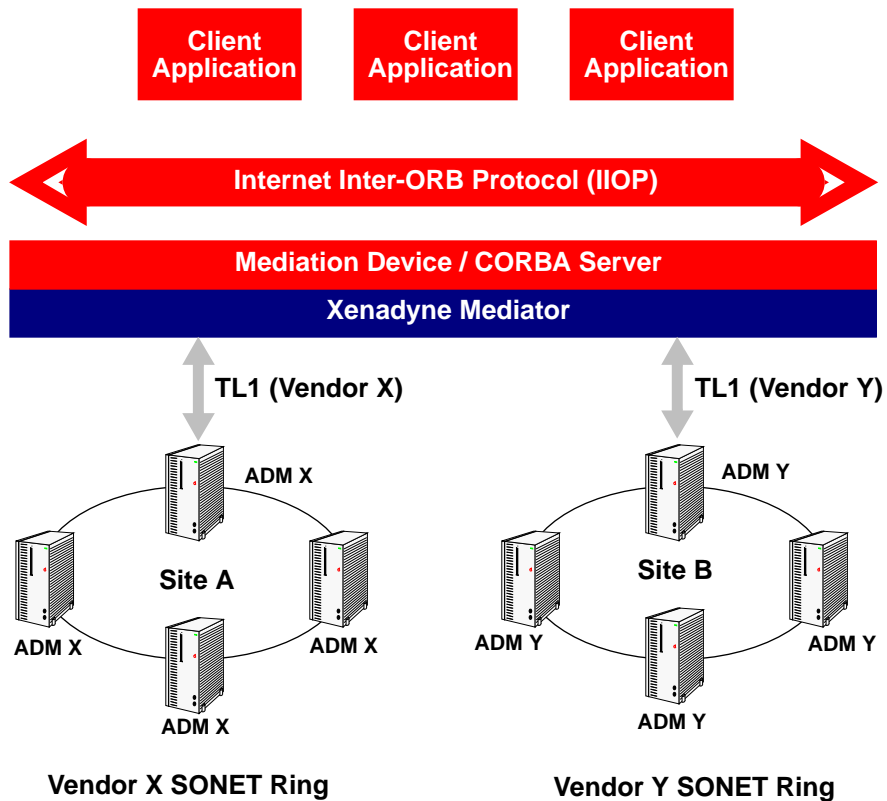


Figure 1-5. Mediation in a CORBA Environment

Xenadyne Mediator

Product Architecture

Xenadyne Mediator is an easy-to-use software development kit for building high-performance mediation components. The product provides all the tools necessary for integrating mediation components with network management systems (NMS), element management systems (EMS), agents, and service-level applications. Mediation components are required for deploying heterogeneous telecommunications and enterprise networks, which consist of multiple network element (NE) types using different protocols from a variety of vendors. Xenadyne Mediator supports heterogeneous networks by normalizing management protocol data units (PDUs) and providing that management information at the Application Programming Interface (API) in a protocol-neutral representation.

The various components that make up Xenadyne Mediator are illustrated in Figure 2-1.

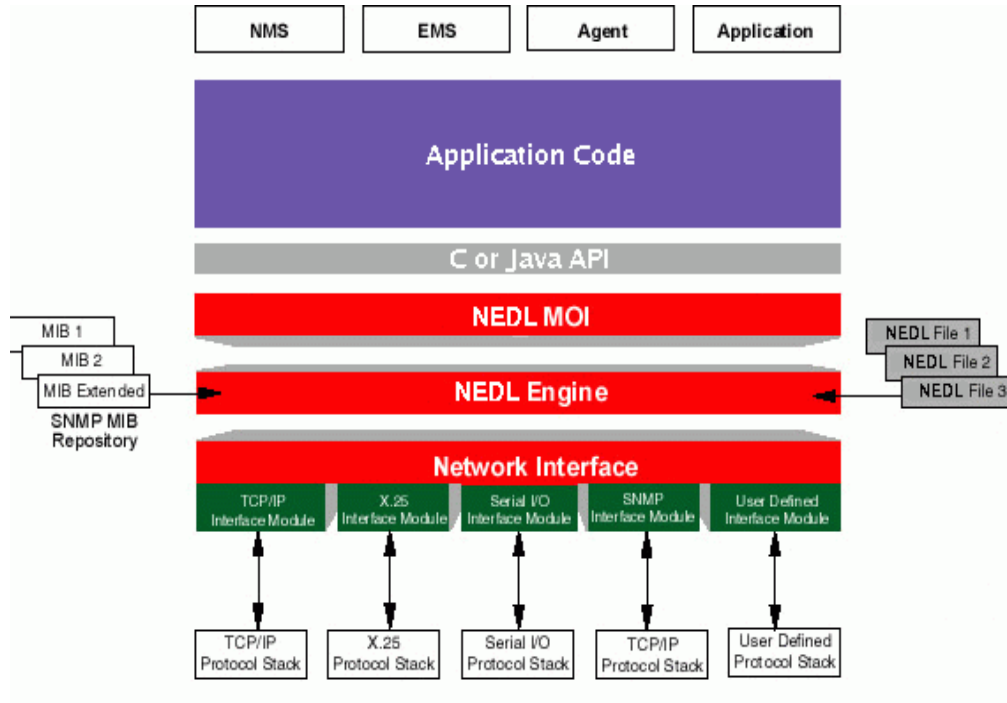


Figure 2-1. Xenadyne Mediator Architecture

The components are described in the following subsections.

Mediator Object Interface (MOI)

The MOI consists of C and Java APIs that provide easy access to and context information of mediated devices. The MOI APIs provide a convenient means for developing protocol and/or object model adapters for use in conjunction with Xenadyne Mediator.

The Mediator provides support for multi-threaded applications. All MOI APIs are thread safe. Multi-threading offers the following benefits:

- Higher throughput for high-volume traffic
- Scalability for multi-processor systems
- Compatibility with multi-threaded application architectures and RTOS's.

The C MOI contains the following components:

- Connection API
- Schema API
- SNMP Proxy API

For more information about the C MOI, see *Chapter 5, "Mediator MOI."*

The Java MOI is defined in the package `com.xenadyne.mediator`, and provides equivalent functions to the C MOI.

Mediator Network Interface

NEs communicate with the Mediator Engine through the Mediator Network Interface module, which has built-in support for the following transport protocols:

- TCP/IP
- Serial I/O
- X.25

Mediator includes Network Interface modules for the following application protocols, using one or more of the above transport protocols:

- ASN.1
- HTTP
- SNMP
- Telnet
- QD2
- TL1 Gateway Multiplexor
- Ericsson CSS
- Siemens EMOS

The TL1 Gateway Multiplexor allows multiple Mediator connections to share a common network link to a Gateway network element. The TL1 Gateway Multiplexor works with the

Mediator's built-in X.25, TCP and Serial network interfaces, as well as customer-built NI modules.

In addition, you can add support for any protocol (standard or proprietary) stack of your choice.

Mediator NEDL Engine

The Rosetta Engine handles three types of information that are passed between the application and NE (see Figure 2-2):

- Unsolicited messages (SNMP traps and events, TL1 autonomous messages, ASCII alarms, etc.) sent by the NE are passed to the application through the NEDL Engine.
- Requests sent by the application are passed to the NE through the NEDL Engine.
- Responses (errors, acknowledgments, etc.) are sent by the NE as a result of a Request sent by an application. The NEDL Engine passes Responses to the application.

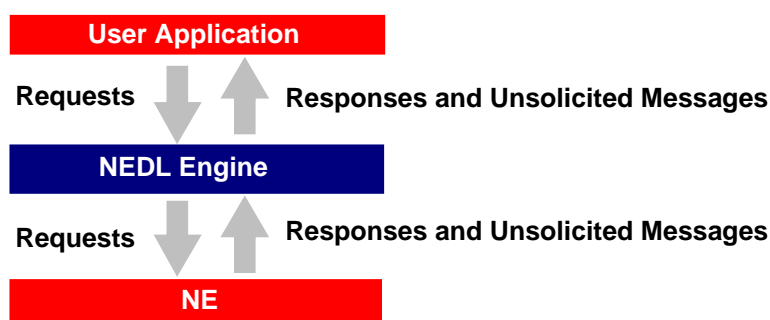


Figure 2-2. Message Types

Xenadyne Mediator is activated when an application opens a connection to an NE and loads a specified NEDL file or SNMP Management Information Base (MIB) to build a NEDL Schema. ***A NEDL Schema is the run-time representation of the NEDL file or SNMP MIB.*** The NEDL Engine performs message transformation between the application and NEs by using the NEDL Schema:

- Requests to be sent to the NE are ***composed*** according to the NE's native protocol and then delivered to the NE via the Mediator Network Interface.
- Messages received from NEs are ***parsed*** and then transformed to a protocol-neutral format and presented to the application via the C or Java APIs. The APIs provide applications with a single interface to the information necessary to monitor and control NEs.

Multiple NEDL files and SNMP MIBs can be loaded into Xenadyne Mediator at run-time, giving simultaneous access to multiple types of equipment from different vendors, using different protocols.

Composing

Manager operations—services such as getting and setting attributes—originate with an application call to the MOI interface and result in a native request message being composed by the Mediator (see Figure 2-3). In the case of a Get call, for example, the process works as follows:

- 1 The attribute “to get” is specified in the Attribute Type parameter in the MOI `moi_get_attribute()` call.
- 2 The appropriate Message is chosen from the Attribute’s MESSAGES list in the schema - in this case, a message with a classification of Atomic Get.
- 3 The Mediator’s NEDL Engine then uses the message pattern to supply the correct format, punctuation, and key words through a substitution and role binding process. The composed request message is then sent to the NE. For more information, refer to *NEDL Author’s Guide* in the product documentation set.

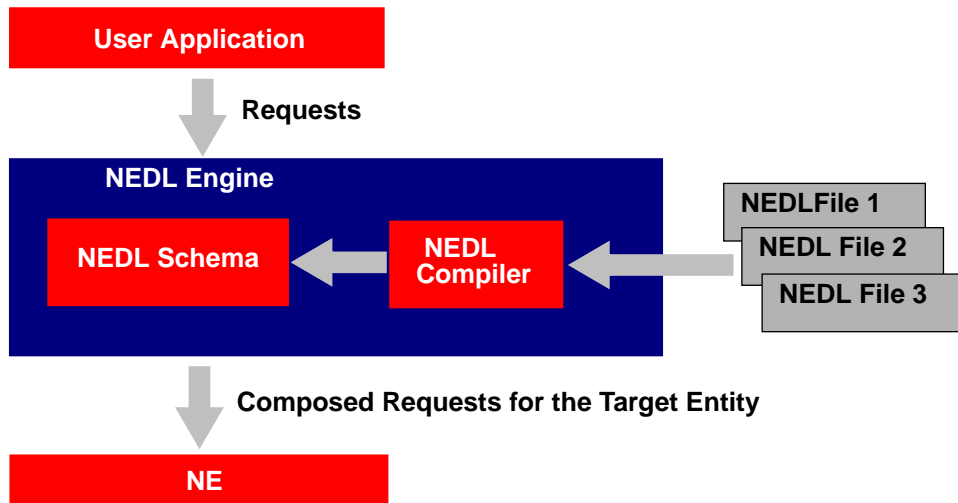


Figure 2-3. Composing

Parsing

Unsolicited messages and responses in the NE's native protocol need to be presented to the management application in a protocol-neutral format. This process is called parsing. In the case of a TL1 Autonomous message, for example, the process works as follows:

- 1 The NE issues an Autonomous message.
- 2 The Mediator's NEDL Engine passes the native message through an efficient parsing network that is generated from the MESSAGE syntax definitions in the NEDL file. This matches the native message text to a NEDL message definition, and generates the corresponding parse tree.
- 3 The NEDL Engine binds the ROLES of the matching NEDL message to the nodes of the message parse tree.
- 4 Each parameter value is MAPPED to a protocol-neutral value, if applicable.
- 5 The NEDL Engine constructs the AVA (Attribute Value Assertion) or OSA (Operational Status Assertion) data structures for the APIs.

For more information about the AVA and OSA data structures, refer to *Chapter 3, "Mediator NEDL Information Model."*

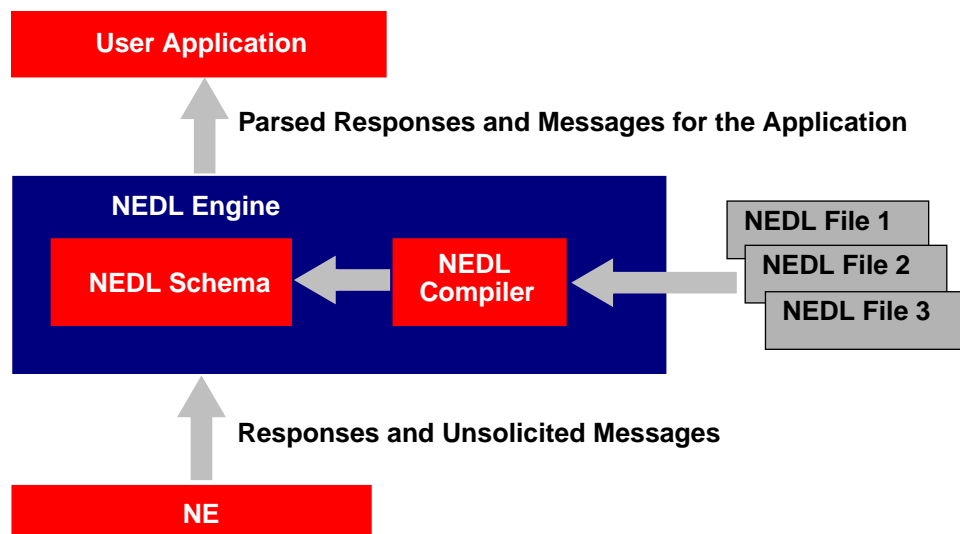


Figure 2-4. Parsing

NEDL Files

A NEDL file contains the description of an NE in terms of its resources and the operations that can be performed on those resources. Xenadyne provides NEDL files for many SONET/SDH ADM (add/drop multiplexers), WDM (wave division multiplexers), and DCC (digital cross-connect) systems, as well as the Bellcore generic message set as specified in Bellcore GR-833 and GR-834.

You can use the pre-existing NEDL files as provided, or you can edit them to create your own NEDL file that describes your specific NE. The collection of NEDL files is called the NEDL Library. Additional information about NEDL files and the NEDL Library is provided later in this document.

SNMP MIBs

In addition to NEDL files, Xenadyne Mediator also uses SNMP MIBs for management of SNMP-based NEs. SNMP MIBs are used to model the management information of an SNMP NE. Xenadyne Mediator supports both SNMPv1 and SNMPv2 for NEs using a standard or enterprise MIB that conforms to Request for Comments (RFC) 1213.

Integrating Xenadyne Mediator

The 2 basic steps for integrating dynamically configurable mediation using Xenadyne Mediator are described in the following subsections:

Step 1: Create and/or test a NEDL file or SNMP MIB

If you are managing ASCII or binary NEs, you need to create and/or test a NEDL file for each NE type. You can create a NEDL file from scratch, or use one of the existing files shipped with the product. If you are managing SNMP devices, then you need to obtain an SNMP MIB from the device vendors.

For more information about creating and testing NEDL files, refer to *Chapter 3, “Mediator NEDL Information Model,”* and *Chapter 4, “NEDL Development Environment.”*

Step 2: Integrate Xenadyne Mediator Using the C or Java MOI with a Management System or Agent

The APIs in the MOI can be used by developers to write a wide variety of management applications, including fault management, software release discovery, performance monitoring, configuration management, provisioning and testing.

Detailed information about integrating applications with Xenadyne Mediator using the Rosetta MOI can be found in *Chapter 5, “Mediator MOI.”*

Mediator NEDL Information Model

Overview

The main purpose of a management information model is to give structure and meaning to the management information exchanged externally by communications protocols. In order to accomplish this, the management information model uses the concept of managed objects. A managed object is an abstract representation of a network element (NE), and is characterized by its attributes, the operations that can be performed on the NE as a whole or on its attributes, and the autonomous messages (notifications) that are generated by the NE.

The Mediator's NEDL Information Model represents an NE as a collection of manageable resources. The NE itself comprises the top-level resource and contains lower-level resources. These resources may contain resources of their own, to as low a level as necessary to adequately describe the entire NE. Each resource, regardless of its level, contains any number of attributes, and the attributes in turn have values (see Figure 3-1). The NEDL Information Model defines a set of operations that can be performed on the attributes. More information about these operations is provided later in this chapter.

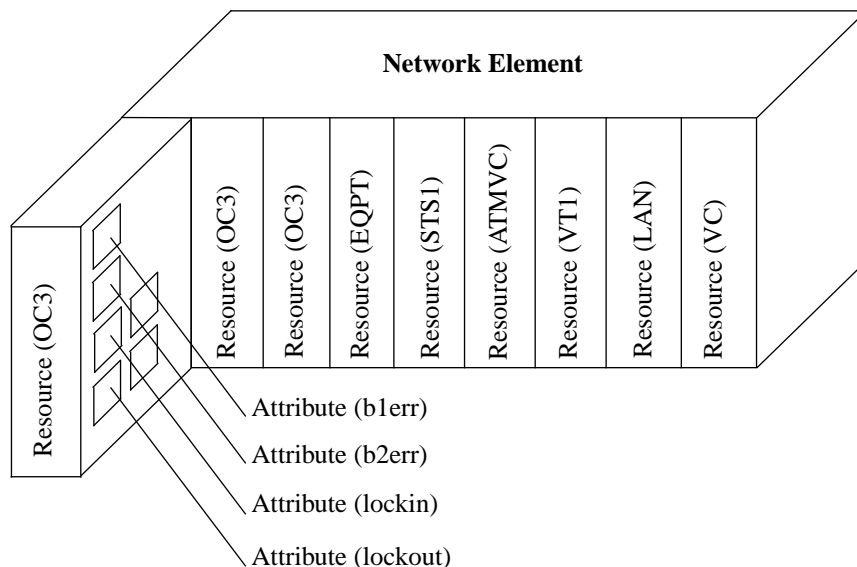


Figure 3-1. An NE as a Collection of Managed Resources and Attributes

In an SNMP (Simple Network Management Protocol) or CMIP (Common Management Information Protocol) environment, NEs are modeled with MIBs (Management Information

Base). In an ASCII/Binary environment, NEs are modeled with NEDL files and the NEDL Information Model.

The NEDL Information Model allows you to create a protocol-neutral model of any NE. This model is described in a NEDL file, which can be created and modified using a text editor. The information needed to build this model can be obtained from the NE's native information management model (for example, SNMP or CMIP MIBs), if one exists. Additionally, the NEDL Information Model can also model an NE that does not have an explicit information model associated with it (for example, an ASCII device).

Defining an NE's Resources and Attributes

As mentioned earlier, the NE itself comprises the top-level resource. In the NEDL file, this special top-level resource is called NETWORKELEMENT. There can be only one instance of the NETWORKELEMENT resource for any specific NE, and, as is the case with all resources, it can have any number of attributes (see Figure 3-2). This concept is similar to the `system` group in an SNMP MIB.

Attribute Names	aat	adt	conf1	conf2	conf3	. . .
Attribute Values						

Figure 3-2. A NETWORKELEMENT and its Attributes

In this example, the NETWORKELEMENT is a SONET add/drop multiplexer (ADM), and its attributes are aat (alarm activate time), adt (alarm deactivate time), conf1 (signal type of Group 1), conf2 (signal type of Group 2), conf3 (signal type of Group 3), etc.

The resources contained within NETWORKELEMENT are defined according to the specific type of NE. For example, a SONET ADM would have resources such as an OC3 (optical carrier—level 3 facility), OC12 (optical carrier—level 12 facility), power supplies, amplifiers, etc. Each resource (for example, an OC3 card) has its own set of attributes (for example: b1err (B1 error data generated), b2err (B2 error data generated), lockin, lockout, etc.). Additionally, there can be multiple instances of each type of resource. This is exemplified in Figure 3-3.

		Resource Type (OC3)							
Attribute Names		b1err	b2err	lockin	lockout	lof	lop	los	. . .
	1								
	2								
	3								
	⋮								
	⋮								

Figure 3-3. Multiple Instances of a Resource Type

Different resource types can have the same attributes in a NEDL file. For example, both OC3 and OC12 cards can have a "LOS" (Loss Of Signal) attribute. A specific instance of an attribute is identified by the following:

- NEID (network element ID)
- Resource type name
- Resource instance ID
- Attribute type name

For example, consider the attribute instance shaded in the table in Figure 3-3. The NEID would be name of the NE, the resource type name is "OC3" (equivalent to the entire table), the resource instance ID is "2" (equivalent to a specific table row), and the attribute type name is "lop" (loss of pointer on STS1). This concept is similar to the notion of an Object Identifier (OID) in an SNMP MIB.

Note: The NEID should not be confused with NETWORKELEMENT. The NEID is generated by the NE and is included in the messages sent by the NE. You define the NETWORKELEMENT in the NEDL file, which is then used by the NEDL Engine to compose messages sent to the NE and parse messages received from the NE.

Figure 3-4 summarizes how an NE is modeled with the NEDL information model.

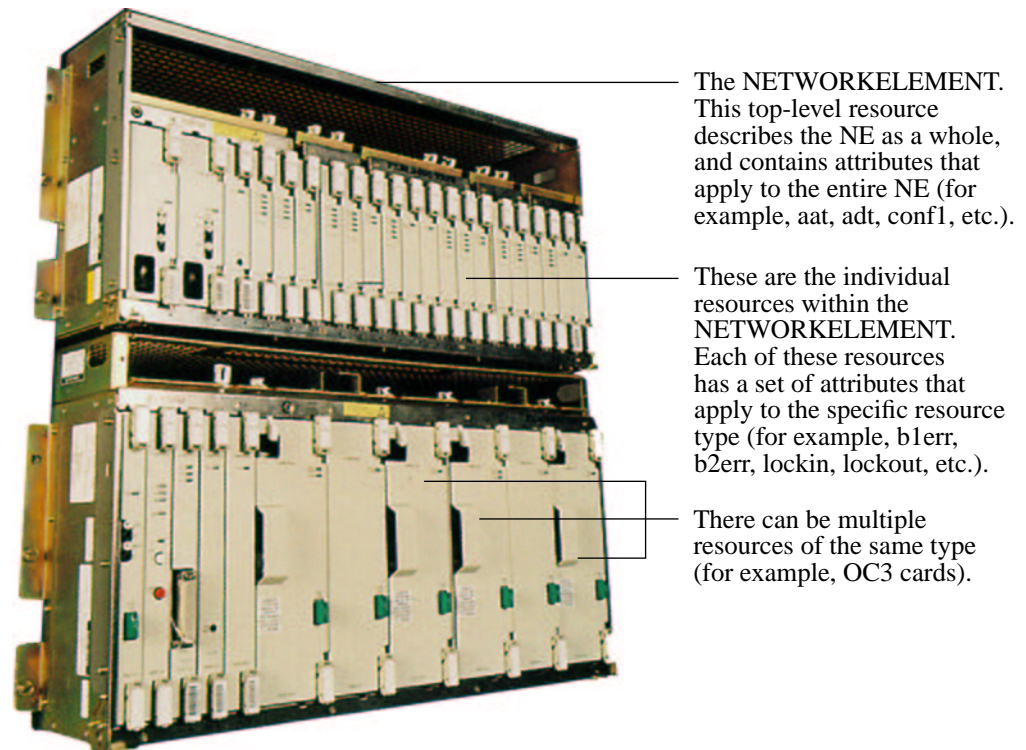


Figure 3-4. A Typical SONET Device and its Resources and Attributes

Mapping Multiple NE Types

The NEDL information model allows you to map similar NEs (for example, ADMs from different vendors) to a generic information model. For example, an OC3 port may be called “OC3,” “OC-3,” or “OC_3,” depending on the NE. In the NEDL information model, all of these specific resource type names are mapped to a single name, such as “OC3.” The same applies for attribute names. The application developer need only deal with the generic information model through the Mediator’s MOI.

Specifying Operations and Access Policy

The NEDL information model allows you to specify operations (Get, Set, etc.) that can be performed on various attributes by resource type. This is accomplished by associating protocol-specific messages that are supported by the NE to perform the following operations:

- Atomic Get, Multi Get, and Bulk Get.
- Atomic Set and Multi Set
- Unsolicited messages (events, alarms, etc.) sent by the NE
- Action requests sent to the resources

By virtue of the operations that you specify, you can determine the access policy for managing the NE. In addition to limiting the operations, the NEDL Information Model allows you to further refine the access policy by limiting the NE’s resources and attributes in the Rosetta file. Therefore, a Rosetta file should be constructed according to the management needs for the specific application.

Note: A craft terminal will always allow full access to the native agent.

NEDL Files and the NEDL Language

As mentioned in the previous section, NEDL files can be built according to the management needs of the specific NE. This is the “top-down” approach for designing a NEDL file: determining the NE’s resources and attributes first, then defining the messages. It is also possible to create a NEDL file from the “bottom-up,” meaning that you define the NE’s messages using the vendor documentation, then define the NE’s resource and attributes.

It is recommended that you design your NEDL files using the top-down approach. One scenario where this would prove useful is if two people are working concurrently on separate NEDL files for two NEs from different vendors. Rather than develop two separate NEDL files independently from each other, it would be easier for both individuals to first agree on the manageable resources and their attributes, and then define the NE’s specific messages.

NEDL File Constructs

A NEDL file contains a series of constructs that define and classify the information within an NE's messages. There are three types of constructs:

- **NE Description**—Describe the NE itself, device-specific numbering requirements for messages, and the NEDL file. These constructs are mandatory, and one and only one of each construct must appear in every NEDL file. The NE description constructs are **IDENTITY**, **MSGID**, and **NETWORKELEMENT**. These and all the other constructs are described in more detail later in this section.
- **Message**—Define the structure and content of the messages sent from the NE to the management application and from the management application to the NE. A NEDL file can have any number of message constructs. The message constructs are **MESSAGE** and **BLOCK**.
- **Managed Resources and Attributes**—Define the resources within the NE, such as an interface card, and the attributes of that resource. The managed resource and attribute constructs are **RESOURCE** and **ATTRIBUTE**.

Identifying the NEDL File and the NE

The **IDENTITY** construct identifies the NEDL file, its definer, the source protocol document, and other useful information for version and document control.

The **MSGID** construct defines the characteristics of the numbering scheme for the NE's messages and describes the format of the messages. Additionally, this construct allows you to specify the message terminator.

The **NETWORKELEMENT** construct is a special **RESOURCE** construct representing the attributes and actions relating to the entire NE. Recall the discussion earlier in this chapter about **NETWORKELEMENT** as the top-level resource for the NE—the definition of this resource is defined in the **NETWORKELEMENT** construct.

A sample **NETWORKELEMENT** construct as it appears in the Rosetta file is shown below:

```
NETWORKELEMENT {
  DESCRIPTION "FT-2000 V5.0 OC48"
  MESSAGES { act-user, canc-user, rtrv-hdr, rtrv-sys,
             rtrv-alm, rtrv-cond, init-reg, dlt-user-secu,
             rept-sw, ed-dat, ed-user-secu, ent-sys,
             rtrv-eqpt, rtrv-map, set-sid, upd-sys,
             rtrv-map-network, rtrv-map-ring, rtrv-attr-env,
             set-attr-env
          }
  ATTRIBUTES { notificationOnOff,
              crossConnectStatus,
              neName,
              AlarmDelay,
              ClearDelay,
          }
}
```

Notice the variety of messages and attributes that exist in this NETWORKELEMENT definition. Each message and attribute is defined in detail later in the NEDL file; each message is defined in a MESSAGE construct, and each attribute is defined in an ATTRIBUTE construct. The MESSAGE and ATTRIBUTE constructs are covered in more detail later in this chapter. For now, it is important to remember that these messages and attributes apply only to the entire NE, as NETWORKELEMENT is the top-level resource.

For more information, see Chapter 4, “NEDL Development Environment.”

Identifying the NE’s Resources and Attributes

Each RESOURCE construct in a NEDL file defines a manageable entity (resource) in the NE. For example:

- A physical entity, such as a switch, relay, or cross-connect
- A format or concept for data, such as a schedule, profile, or trouble report
- A dynamic entity, such as a connection or a virtual circuit
- The NE itself

Note: The NE itself is defined by a special RESOURCE construct called NETWORKELEMENT. Do not enter the information describing the entire NE in a RESOURCE construct.

Each ATTRIBUTE construct defines an attribute listed in a RESOURCE construct. The information in this construct is used in operations, management, or provisioning of the NE.

Consider the following sample RESOURCE construct:

```
RESOURCE EC-1 {
  DESCRIPTION "The EC-1 resource"
  MESSAGES { ent-attr, set-attr, rept-evt, opr-lpbk-ecl,
             rls-lpbk-ecl, rtrv-alm, rtrv-attr, rtrv-cond,
             rtrv-pm, rtrv-states, rtrv-th, rtrv-attr-env
  }
  ATTRIBUTES { NotificationCode, SignalFailureThreshold,
              CVL, ESL, SEFS, SESL, UASL,
              T-CVL, T-ESL, T-SEFS, T-SESL, T-UASL
  }
}
```

Notice the attributes and messages listed in this resource definition. Unlike the messages and attributes listed in the NETWORKELEMENT construct, these messages and attributes apply only to the EC-1 resource.

Each message and attribute is defined in its own construct. For example, the NotificationCode and CVL attributes are defined in their own ATTRIBUTE constructs, as follows:

```
STATE NotificationCode {
    DESCRIPTION "Alarm priority indication"
    ATTRIBUTETYPE "NTFCNCDE"
    ATTRIBUTEVALUE RANGE { "CR", "MJ", "MN", "NA" }
    MESSAGES { ent-attr, set-attr }
}

COUNT CVL {
    DESCRIPTION "Coding Violation Count - Line"
    ATTRIBUTETYPE "CVL"
    MESSAGES {rtrv-pm}
}
```

Notice the keywords STATE and COUNT preceding the attribute names NotificationCode and CVL, respectively. These keywords identify the attribute type. The NEDL Language defines seven types of attributes, which are listed and described briefly in Table 3-1.

Table 3-1. Types of Attributes

Attribute Type	Description
ALARM	Indicates the existence of a fault in a resource or NE.
COUNT	Defines a counter that represents an underlying counting process.
GAUGE	Reflects a dynamic quantity, such as the number of connections currently open, or the rate of change for a counter.
OPAQUE	An attribute whose value is encoded.
STATE	An attribute with an enumerated set of values.
STRING	An attribute value consisting of printable text.
THRESHOLD	Describes a notification level or a warning level.

Messages are described in the next section.

Defining Messages and Blocks

Two of the basic functions required to manage remote NEs are encoding request messages to be sent to the NE, and decoding the responses. For ASCII and binary management protocols, it is necessary to specify the format of the messages to enable encoding and decoding. The NEDL language provides a rich set of primitives to efficiently specify message structures.

Defining Blocks

Whether the protocol is ASCII or binary, all NEDL messages can be decomposed into component blocks, which are defined in BLOCK constructs. BLOCKs are message components that can be shared among message definitions, and can also be used to compose higher-level blocks.

The sample BLOCK construct below defines the TL1 header, which is common to many TL1 messages. It contains references to the “sid,” “date,” and “time” BLOCKs:

```
BLOCK tll_header {  
    PATTERN { "<sid> <date> <time>" }  
    EXAMPLE { "LAX007 95-09-15 12:14:26" }  
}
```

For ASCII protocols, all BLOCKs must ultimately decompose into REGEXPs (regular expressions) that enable the Mediator’s NEDL Engine to parse the BLOCK when it appears in a message. In the example above, a “date” consists of numerals separated by dashes. The BLOCK definition for “date” could express this as follows:

```
BLOCK date {  
    REGEXP { "[-0-9]+" }  
    EXAMPLE { "95-09-15" }  
}
```

For binary protocols, BLOCKs decompose into sets of bits and bytes. Consider a variable-length string field in a binary message that consists of a length byte, followed by a character array of the given length. To specify this in Rosetta, the length field is first specified as a fixed one-byte BLOCK. The character is defined as one byte also. The variable string is defined by applying the array operator [] to <character>, as shown below:

```
BLOCK length {  
    BYTES { 1 }  
}  
  
BLOCK character {  
    BYTES { 1 }  
}  
  
BLOCK var_string {  
    BYTES { "<length><character>[length]" }  
    EXAMPLE { "\031a string of 25 characters" }  
}
```

Defining Messages

Now let's examine one of the messages from the ATTRIBUTE example in the previous section. Recall that the NotificationCode attribute contains two messages: ent-attr and set-attr. The set-attr message is defined as follows:

```
MESSAGE set-attr {
  DESCRIPTION "Set the notification code associated with
              incoming signal failure events"
  REQUEST {
    PATTERN { "SET-ATTR-<modifier>:<tid>:<aid>:<ctag>"
              " ::<ntfcncde>,,,,<t11_req_term>"
    }
    ROLES {
      RESOURCETYPE = modifier
      RESOURCEID   = aid
      OTHER        = ntfcncde RANGE { "CR", "MJ", "MN", "NA" }
    }
    EXAMPLE {
      "SET-ATTR-T3:ATT-DDM-2000:1s-5a-2:123456::CR,,,,;"
    }
  }
  RESPONSE { PATTERN { "<t11_success_response>" }}
  ERROR { PATTERN { "<t11_error_response>" }}
  ACK { PATTERN { "<acknowledge_message>" }}
}
```

Notice the various BLOCK definitions referenced in this message definition: "modifier," "tid," "aid," "ctag," etc. Each BLOCK is defined elsewhere in the NEDL file.

The DESCRIPTION clause contains a brief description of this message, and the ERROR clause defines the error message returned from the NE. In this case, the error is defined in the "t11_error_response" BLOCK definition.

The operations associated with the set-attr message are defined in the REQUEST, RESPONSE, and ACK clauses. This means that the set-attr message can be sent by the management application as a request or by the NE as a response or acknowledgment. The format of the message for each operation is defined in the PATTERN clause, and an example message is provided in the EXAMPLE clause.

Note: The EXAMPLE inside the RESPONSE clause can be used by the Simulator to imitate the behavior of an NE. For more information, refer to *NEDL Author's Guide* in the product documentation.

Notice the ROLES clause inside the REQUEST clause. NEDL roles are described in more detail in the next section.

Defining Roles

Each parameter in a message is included in the protocol message for a specific management purpose, or “role.” The data in each parameter can be classified into a NEDL role that represents its purpose. This role is defined in a ROLES clause in the NEDL file. A ROLES clause can be defined inside a MESSAGE or BLOCK construct.

Every parameter must fit into a NEDL role. When you assign a parameter to a role using the ROLES clause, you are explicitly categorizing the parameter’s purpose. The role defines the structure and purpose of that message parameter for the Mediator.

The service or operation provided by a message can be determined solely from the type of parameters it supports. In other words, you can determine the purpose of a message by looking at the roles into which the message’s parameters have been assigned.

For example, in a TL1 protocol message, the “tid” parameter is the terminal ID. In a MESSAGE construct, the “tid” parameter would be assigned into the NEID role to identify the NE by name. In the previous example of the `rtrv-pm` message, you can see that the “modifier” parameter is assigned to the RESOURCETYPE role, and the “aid” parameter is assigned to the RESOURCEID role, and so on.

Roles Affecting the Message as a Whole

NEDL defines many roles that can be assigned to the parameters in each message. The roles listed below in Table 3-2 affect the message as a whole.

Table 3-2. Roles Affecting the Message as a Whole

Role	Description
MESSAGEDATE	The date the message was sent by the NE.
MESSAGEID	The unique ID of the message.
MESSAGETIME	The time the message was sent by the NE.
MORETOCOME	The presence of this role indicates that the message contains acknowledgments or multiple responses.

Roles Affecting the NE as a Whole

The roles listed below in Table 3-3 affect the NE as a whole.

Table 3-3. Roles Affecting the NE as a Whole

Role	Description
NEID	The unique ID of the network element.
RESOURCEID	The unique ID of a specific resource.
RESOURCETYPE	The resource type.

Roles Specifying Whether the Message Should be Stored as an AVA, or OSA

Messages that are received from the NE (responses and autonomous messages) are parsed and transformed to one or more instances of the following data structures:

- AVA (Attribute Value Assertion)
- OSA (Operational Status Assertion)

Generally, queries on attributes result in response messages containing AVAs. Request messages that invoke resource actions do not refer to attributes, and therefore will generally produce OSAs.

In order to determine the data structure format in which the message should be stored, the Mediator looks for specific trigger ROLES in the message parse tree. Each binding of ATTRIBUTEVALUE or OPSTATUS in the parse tree generates an AVA or OSA respectively, as shown below in Table 3-4.

Table 3-4. Trigger Roles Generate AVAs, ASAs or OSAs.

Role	Description
ATTRIBUTEVALUE	The value that is being asserted in a message for an attribute. This role triggers generation of an AVA.
OPSTATUS	The operational status being asserted in the message. This role triggers generation of an OSA.

The roles listed below in Table 3-5 provide additional information in the AVA or OSA.

Table 3-5. Roles Providing Information about an AVA, ASA or OSA.

Role	Description
ATTRIBUTEDATE	The date on which an ATTRIBUTEVALUE or ALARMSTATUS was taken on the NE.
ATTRIBUTETIME	The time when an ATTRIBUTEVALUE or ALARMSTATUS was taken on the NE.
ATTRIBUTETYPE	The protocol name for the attribute or alarm being monitored in the operation.

The OTHER Role

Message parameters that are required by the management application but which do not fit into a Rosetta role (for example, a parameter containing the time period over which the threshold is measured) are defined by the OTHER role. These “other” parameters can all be assigned into the OTHER role, even when there is more than one such parameter in the MESSAGE or BLOCK definition. By using the OTHER role, these parameters are made accessible to the API of the management application.

For more information about roles, refer to *NEDL Author's Guide* in the product documentation.

NEDL Development Environment

The NEDL development environment consists of a collection of modules that allow you to create and test a NEDL file.

The following modules comprise the NEDL development environment:

- NEDL Library
- Nedl Tools
- Test Scripting Language (TSL)
- Network Element Simulator

Each module is described in detail throughout the remainder of this chapter.

Building NEDL Files

NEDL files can be edited using any text editor. Usually it is best to begin with one of the NEDL files in the NEDL Library, customizing it to suit your device and application. NEDL files can be tested with the Test Scripting Language (TSL), which is discussed in greater detail later in this chapter. Xenadyne Mediator includes the following tools to assist with NEDL development:

- **nedl_check** - compiles NEDL files, reporting syntax and semantic errors.
- **nedl_schema** - produce a schema summary report from a NEDL file.
- **nedl_2html** - produce an HTML summary of a NEDL file.
- **nedl_2gdm** - convert a NEDL object model to GDMO MIB.
- **nedl_2mib** - convert a NEDL object model to an SNMP MIB.

Detailed information about creating and testing a NEDL file is provided in *NEDL Author's Guide* in the product documentation set.

Generating Reports

The **nedl_check** utility:

- Loads and compiles the NEDL file (or SNMP MIB), reporting any syntactic errors.
- Ensures that all constructs defined in the NEDL file are referred to within the file and contain the appropriate clauses.

- Checks that references actually point to defined constructs, and that those references are consistent with the scope of the message definition.
- Checks each message against its CLASSIFY clause to be sure that the message definition and ROLE bindings conform to the indicated operation.
- Checks each EXAMPLE against the corresponding message or block pattern, to ensure that the example parses correctly. Parsing details appear in the nedl.details report, which we describe below.

The nedl_check utility generates two reports (output files) that you can use for debugging your NEDL file: nedl.classify and nedl.details.

The nedl_schema utility loads the specified NEDL file and generates a schema report listing the messages, resources and attributes defined in the file.

These reports are described in the following subsections.

Message Classification Reports (nedl.classify)

Message classification reports identify the type of operation associated with each MESSAGE defined in the NEDL file. The following is an excerpt from the nedl.classify file that is created when nedl_check is run against the ft2000.nedl file:

```
act-user          NE_Action
alw-msg           Atomic_Set
canc-user         NE_Action
dlt-crs           Resource_Action
dlt-user-secu    NE_Action
ed-dat           NE_Action
ed-pid           NE_Action
ed-user-secu     NE_Action
ent-attr         Resource_Action
ent-crs          Atomic_Set
```

The name of each MESSAGE appears in the left column, while the operation type appears in the right column. If there are problems in the NEDL file, the message classification report contains error and warning messages about these problems. Some common problems might be:

- Repeated BLOCK definitions
- Circular BLOCK references
- Missing delimiters (for example, <>, [], {}, (), " ", etc.)
- Misspelled reserved words and keywords (NEDL reserved words must be all uppercase letters)
- Illegal nesting

Detailed Message and Pattern Block Reports (nedl.details)

Detailed message and pattern block reports contain lists showing how PATTERNS or REGEXPs in the file are matched to EXAMPLEs. Below is an excerpt from the

nedl.details file that is created when nedl_check is run against the ft2000.nedl file:

```
Message "set-attr" (defined on line 3360 in ft2000.nedl)
  Block "set-attr_request"
    This request matches the pattern for "set-attr_request"
    For the example "SET-ATTR-T3:ATT-DDM-2000:ls-5a-
2:123456::CR,,,,;"
    The roles that get bound are:
      role OTHER "ntfcncde" = "CR"
      role RESOURCEID = "ls-5a-2"
      role RESOURCETYPE = "T3"
      role NEID = "ATT-DDM-2000"
      role RESOURCEID = "ls-5a-2"
      role MESSAGEID = "123456"
    The matching patterns are:
      set-attr_request matches "SET-ATTR-T3:ATT-DDM-2000:ls-
5a-2:123456::CR,,,,;"
      modifier matches "T3"
      tid matches "ATT-DDM-2000"
      clli matches "ATT-DDM-2000"
      aid matches "ls-5a-2"
      aidlayout matches "ls-5a-2"
      ctag matches "123456"
      taglayout matches "123456"
      ntfcncde matches "CR"
```

The EXAMPLE clause is identified in the fourth line of this excerpt. The EXAMPLE clause is used to both illustrate and validate the syntax given the PATTERN clause.

Schema Reports

The **nedl_schema** utility produces schema reports containing the following information:

- Resources and their associated attributes and operations (messages)
- Attributes, their categories, and their allowed values and operations
- Operations and their associated mandatory parameters

Below is an excerpt from the schema that is created when nedl_schema is run against the ft2000.nedl file:

```
RESOURCE Equipment
  DESCRIPTION: "The Equipment resource"
  RESOURCEIDS: "ALL"
  OPERATIONS: rept-evt
               rtrv-alm
               rtrv-cond
               rtrv-eqpt
               rtrv-modifier
```

```
sw-toprotn
sw-towkg
ATTRIBUTES: ModeSwitchProtection
            ModeSwitchToWorking
            equipmenttype

ATTRIBUTE NotificationCode
DESCRIPTION: "Alarm priority indication"
PROTOCOLNAME: NTFNCNDE
TYPE: State
VALUES: "CR" "MJ" "MN" "NA"
OPERATIONS:

ACTION set-attr
DESCRIPTION: "Set the notification code associated with
            incoming signal failure events"
REQUEST OTHER PARAMETERS: ntfncnde { "CR" "MJ" "MN" "NA" }
```

This excerpt shows the `Equipment` resource and its associated operations and attributes, the `NotificationCode` attribute and its allowed values and operations, and the `set-attr` action (operation) and its required parameters.

NEDL Library

Xenadyne provides NEDL files for many SONET/SDH ADM (add/drop multiplexers), WDM (wave division multiplexers), and DCC (digital cross-connect) systems, as well as the Bellcore generic message set as specified in Bellcore GR-833 and GR-834. This collection of pre-existing NEDL files is called the NEDL Library.

You can use the pre-existing NEDL files as provided, or you can edit them to create your own NEDL files that describe the specific network elements in your network. You can cut-and-paste parts of an existing NEDL file to create a new NEDL file.

Xenadyne is constantly creating new NEDL files. For updates, visit the Xenadyne web site at <http://www.xenadyne.com>.

Test Scripting Language

The Test Scripting Language (TSL) provides a command line interface to the Mediator MOI and enables you to test a NEDL file against a real device to ensure that requests are processed correctly, and that responses and autonomous messages match what you specify in the NEDL file. Figure 4-1 illustrates the TSL environment.

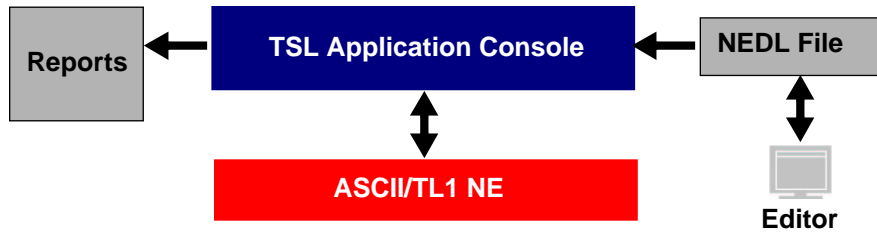


Figure 4-1. ASCII/TL1 Device Test Environment

You should always test your NEDL files because:

- An error may have been made when interpreting the specifications for the device
- The specifications for the device may be out of date

TSL Commands

TSL consists of a set of commands and options, which can be entered interactively or by running script files. The TSL commands enable you to:

- Connect and disconnect from multiple NEs
- Retrieve or modify attribute values
- Perform an action on a resource or on the NE itself
- Set timeout values for responses
- Query the schema to determine which operations can be performed

Table 4-1 lists the TSL commands and provides a brief description for each command.

Table 4-1. TSL Commands

Function	Description
action	Perform an Action on a resource or NE.
auto	Simulate an Autonomous event message from the NE.

Table 4-1. TSL Commands (continued)

Function	Description
create	Send a Resource Create request to the NE
bget	Perform a Bulk Get operation on a resource or NE.
change	Change the NEDL file associated with the connection, allowing you to exercise various NEDL files against the NE without having to disconnect and reconnect with the NE.
connect	Assign an NE handle and connect to the NE.
debug	Set the debugging options.
delete	Send a Delete Resource request to the NE.
diagnose	Check through the NE's schema and produce a list of applicable operations.
disconnect	Terminate the connection with the NE.
display	Browse information in the schema.
echo	Write arguments to stdout.
exit	Terminate TSL.
expect	Specify the result that the next operation should return.
get	Retrieve the value of a specific attribute.
handle	Change to another connected NE with the specified NE handle.
help	Show the command line format for a specified command.
mget	Get multiple attribute values from a specific resource of the NE.
mset	Set multiple attribute values on a specific resource of the NE.
reload	Reload a NEDL file—load a revised NEDL file. This command enables you to test changes to a NEDL file without having to disconnect and reconnect with the NE. All connections using the NEDL file are updated.
send	Send a raw (pass-through) message.
set	Modify the value of a specific attribute.
status	Show all open connections and the current date and time.

Table 4-1. TSL Commands (continued)

Function	Description
timeout	Globally set the maximum time that TSL should wait for a response from the NE after sending a request.
wait	Signal the Simulator to generate an autonomous message.

Diagnostic Reports

Use the `diagnose` command in TSL to generate a report listing all of the allowable operations that can be performed on the specified NE. By default, the report can be found in the `diagnose.tsl` file, which is generated when you run the `diagnose` command. You can, however, specify a file of your choice or print the report to `stdout`.

Below is an excerpt from the `diagnose.tsl` file that is created when the `diagnose` command is run against the `ddm2000.ned1` file:

```
action NETWORKELEMENT set-sid sid=<sid>
bget NETWORKELEMENT rtrv-attr-alm
bget NETWORKELEMENT rtrv-cond
bget NETWORKELEMENT rtrv-pmsched
bget NETWORKELEMENT rtrv-sys
bget NETWORKELEMENT rtrv-ulsdcc aid=ALL
set NETWORKELEMENT AlarmGatewayNetwork YES
set NETWORKELEMENT AlarmGatewayNetwork NO
set NETWORKELEMENT DCCMode DISTINCT
set NETWORKELEMENT DCCMode IDENTICAL
```

The operation (`action`, `bget`, `set`, etc.) appears in the far left column, followed by the name of the resource on which the operation can be performed, the name of the message, and any OTHER role defined in the message.

In the first line of this excerpt, we can see that the `set-sid` message performs an Action operation. The “sid” in angled brackets (<>) means that you must manually supply this value. The `rtrv-ulsdcc` message in the sixth line of this excerpt performs a Bulk Get operation, and the OTHER role “aid” is defined as “ALL.”

The first two Set operations listed indicate that the `AlarmGatewayNetwork` attribute for the `NETWORKELEMENT` resource may be set to either “YES” or “NO.” The last two Set operations indicate that the `DCCMode` attribute may be set to either “DISTINCT” or “IDENTICAL.”

For more information about TSL, refer to *NEDL Author’s Guide* in the product documentation.

Simulated Connections

You can exercise your NEDL file in TSL using the built-in simulator network interface. This network interface internally generates responses and autonomous messages using the EXAMPLEs in your NEDL file, without actually connecting to a remote system. To use the Simulator, specify “sim” as the network interface in the CONNECT command. The Simulator disregards the address string. For example:

```
tsl> 1 connect myfile.nedl DEFAULT-NE sim foo
```

Simulators

Xenadyne Mediator includes a standalone network element simulator for ASCII management protocols like TL1. The simulator accepts connections over TCP/IP, and can simulate any number of network elements and network element types concurrently.

The **Simulator** uses a NEDL file to simulate the messages and behavior of the corresponding NE type. In order to test response messages, make sure that an EXAMPLE clause exists in the RESPONSE sub-block of the NEDL file. When the Simulator receives a request, it sends the EXAMPLE back to the source of the request. Similarly, an EXAMPLE clause must exist in order to test unsolicited/autonomous messages sent by the NE.

For more information about the Simulator, refer to *NEDL Author's Guide* in the product documentation.

The Mediator product set also includes the **ADM2000** simulator, which simulates a generic SONET add-drop multiplexor using the TL1 protocol. The ADM2000 simulator does not use a NEDL file, as all of its behaviors and responses are coded in, but it supports that management features defined in the **adm2000.nedl** NEDL file. This simulator and NEDL file are used in many of the demonstrations and examples.

The ADM2000 simulator is implemented using the Xenadyne Mediator ASCII Agent Framework, and full source code for the ADM2000 is provided, so that you can extend and customize its behavior.

Mediator MOI

Applications such as proxy agents, management protocol adaptors, etc., are linked to the Mediator's NEDL Engine through protocol-neutral C or Java APIs that comprise the Mediator MOI (Mediator Object Interface). The Mediator's MOI consists of:

- Connection API
- Operations API (including service for raw or pass-through messages)
- Schema API

This chapter provides an overview of the features and functionality available in the Mediator MOI APIs for the C programming language. For detailed information on these APIs, refer to the following books in the product documentation set:

- *Mediator C API Reference Manual*
- *Mediator Java API Reference Manual*

Mediator MOI C API

Connection API Functions

The Connection API contains functions that provide connection and connection management information between management applications and NEs. Functionality for the following is supported:

- Initializing the Mediator MOI, NEDL Engine, and Network Interface
- Connecting/disconnection to/from a NE
- Performing local management functions and queries

The main functions in the Connection API are listed in Table 5-1.

Table 5-1. Main Connection API Function

Function	Description
<code>moi_initialize()</code>	Initializes the Mediator MOI and NEDL Engine.
<code>moi_connect()</code>	Connects the Mediator to an NE and returns an NE handle if successful.
<code>moi_disconnect()</code>	Disconnects the Mediator from the NE corresponding to the specified NE handle.
<code>moi_get_neh_*</code>	A group of functions that provide information for a particular connection as identified by an NE handle.

A network element handle (NE handle) is returned by `moi_connect()` each time a connection with the NE is established. An NE handle is the run-time, protocol-neutral information used by the NEDL Engine to identify a specific NE. This is not to be confused with the network element ID (NEID), which is the protocol-specific information used by the NE in its responses or unsolicited messages.

Operations API Functions

The Operations API allows you to perform “operations” using the following message types:

- Get requests
- Set requests
- Actions
- Unsolicited messages (notifications)

The various operations and their corresponding functions are described in Table 5-2.

Table 5-2. Management Operations and Corresponding Functions

Operation	Operations Unit Function(s)	Description
Atomic Get	<code>moi_get_attribute()</code>	Retrieves the attribute value of <i>one</i> attribute in a resource.
Multi Get	<code>moi_get_multi_begin()</code> <code>moi_get_multi_add()</code> <code>moi_get_multi_send()</code>	Build and send a request to retrieve one or more attributes in a resource.
Bulk Get	<code>moi_get_bulk()</code>	Retrieves multiple attributes in the specified resource.
Atomic Set	<code>moi_set_attribute()</code>	Modifies the value of an attribute in a resource.
Bulk Set	<code>moi_set_multi_begin()</code> <code>moi_set_add_add()</code> <code>moi_set_add_send()</code>	Set multiple attributes in the specified resource.
Action	<code>moi_send_action()</code>	Performs some action on a specified resource, but does not explicitly set or retrieve attribute values.
Notify	<code>moi_get_next_event()</code> or <code>moi_get_neh_event()</code>	Receives an unsolicited message from the NE. This message can indicate an alarm, attribute value change, or operational status change. In addition, the raw message is made available to the application.
N/A	<code>moi_send_raw_msg()</code>	Send a raw message to an NE. This allows an application to communicate directly with the NE without involving the Mediator.

Schema API Functions

The Schema API provides read-only access to the NEDL Schema. These functions can be used by the application developer to access the information from the NEDL file or SNMP MIB for a particular NE type. This allows an application to dynamically determine the NE's characteristics (for example, resource types and attribute types). In other words, you can develop dynamically, self-configurable applications that can learn the NE's characteristics during run-time and manage the NEs accordingly.

Some of the main functions in the Schema API are listed in Table 5-3.

Table 5-3. Schema API Functions for Obtaining Information from an NE

Function	Description
<code>moi_sch_resource()</code>	Get a list of all the NE's resources.
<code>moi_sch_attributes()</code>	Get a list of all the NE's attributes.
<code>moi_sch_res_attributes()</code>	Get a list of attributes belonging to the specified resource.
<code>moi_sch_attr_resources()</code>	Get a list of resources containing the specified attribute.
<code>moi_sch_attr_is_*</code>	Group of functions that tells you whether or not an action may be performed on the specified attribute.
<code>moi_sch_operations()</code>	Get a list of all the NE's Operations.

Technical Details and References

This appendix describes the technical details of Xenadyne Mediator and lists the technical references.

Technical Details

The technical details for Xenadyne Mediator are described in the following subsections.

Message Throughput

Xenadyne Mediator can process 5000 alarm messages per second (average 128 bytes per message) on a modern single-processor host. Throughput scales linearly with additional processors.

Network Element Protocols

Xenadyne Mediator supports:

- Proprietary ASCII protocols
- TL1 (Bellcore GR-831)
- SNMP versions 1, 2c and 3.
- Binary (or bitstream) protocols such as QD2 and ASN.1

Network Communications Protocols

Xenadyne Mediator supports basic transport protocols like:

- TCP/IP
- Serial I/O
- SunLink X.25 (Solaris only)
- User-implemented protocols, such as SSL

Mediator also supports higher-level protocols such as

- Telnet
- HTTP

Proxy Agents/Management Protocol Adaptors

Xenadyne Mediator can be integrated with a variety of agent toolkits to develop proxy agents or management protocol adaptors (MPAs) for different management protocols, including:

- SNMP
- CMIP
- CORBA
- XML

Mediator includes an SNMP Proxy Agent API specifically for the purpose of constructing the SNMP Agent portions of the SNMP proxy agent.

Management Application Integration

Mediator can be integrated with management applications written in any of the following languages:

- C
- C++
- Java

NEDL Library

Currently, the NEDL Library includes:

- **Add/Drop Multiplexers**
 - *Lucent*: DDM-2000, FT-2000
 - *Fujitsu*: FLM150, FLM600, FLM2400
 - *Alcatel*: 1603/1612 SM, 1648 SM
 - *Nortel*: S/DMS Transport Node
- **Digital Cross-Connects**
 - *Lucent*: DACS III, DACS IV
- **Wave Division Multiplexers**
 - *Pirelli*: WDM16
- **TL1 Generic Requirements**
 - *Bellcore*: GR-833, GR-834

Supported Platforms

Xenadyne Mediator can run on the following platforms:

- Sun Solaris 2.7, 8 and Solaris 9 on Sparc.
- Sun Solaris 9 on x86
- Red Hat Linux 9.0 on x86.
- Windows NT/2000/XP
- HP-UX 11
- IBM AIX 4.3.3

Memory and Disk Space Requirements

The memory and disk space requirements for Xenadyne Mediator are listed below:

- Minimum memory requirement: 64MB
- Minimum disk space requirement for development: 40MB
- Average disk space requirement for run-time data: 5MB

Technical References

Technical references are listed below:

- **Bellcore GR-831-CORE, OTGR Section 12.1**, Operations Application Messages - Languages for Operations Application Messages, Issue 1, November 1996
- **Request for Comments (RFC) 1156**, Management Information Base for network management of TCP/IP-based internets, May 1990
- **Request for Comments (RFC) 1213**, Management Information Base for network management of TCP/IP-based internets: MIB-II, March 1991