Modeling and Testing Main IPv6 Protocols

Zheying Huang

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presenting in Partial Fulfillment of the Requirements

For the Degree of Master of Applied Science

Concordia University

Montreal, Quebec, Canada

April 2004

Abstract

Modeling and Testing Main IPv6 Protocols

Zheying Huang

This thesis presents the modeling and testing of main network layer protocols in IPv6, which are IPv6, ICMPv6, MLD and RIPng protocols. The purpose of this thesis is to formalize these protocols and to automatically generate test suites for conformance and interoperability testing. The protocols are specified using SDL (Specification and Description Language) and the test suites are described with TTCN (Tree and Tabular Combined Notation). The basic approach of formalizing these protocols applies the principle of SDL: different abstract levels are covered from broad overview down to the detailed design level. The test case suites are generated by test generator-testComposer in ObjectGeode toolset. A brief overview of the four main protocols is given before the general approach. The examples are given following all applied steps.

TABLE OF CONTENTS

Table of Figures

1. Introduction

The term IPv6 stands for "Internet Protocol Version 6". IPv6 is described in Request for Comments (RFC) 2460. It is the "next generation" protocol designed to replace the current version of Internet Protocol, IP Version 4 ("IPv4"). IPv4 has proven remarkably robust, easy to implement, and interoperable with wide range of protocols and applications. However, with the ongoing explosive growth of Internet and Internet service, IPv4 has become outdated for it cannot easily meet the requirements of a multimedia, application-rich environment. IPv6 fixes a number of problems of IPv4, such as addressing, address auto-configuration, security, Quality of Service (QoS), routing, and fragmentation.

IPv6 extends addressing capabilities by increasing IP address from 32-bit to 128-bit. It supports more levels of addressing hierarchy to overcome the limitation of the two-level structure of the IP address in IPv4.

IPv6 also defines both a stateful and stateless address auto-configuration mechanism to simplify configuration of a host. As a result, auto-configuration reduces the time, cost, and complexity of renumbering new IPv6 addresses of existing hosts.

Security is mandatory in IPv6, whereas in IPv4 it is optional. Moreover, IPv4 lacks effective privacy and authentication mechanisms under the application layer. Consequence, IPv4 can cause serious risk in guaranteeing a secure environment. In contrast to IPv4, IPv6 offers security features at router level of the TCP/IP architecture, where they can benefit all TCP/IP application. It provides security encryption, authentication and data integrity safeguards.

IPv6 offers two QoS features: flow labels and traffic classes. The flow label enables the flow's source to identify a logical sequence of packets. The traffic class allows a host to specify a packet priority to permit IPv6 routers to discriminate and favourably execute TCP/IP applications that request faster response time.

Routing in IPv6 is similar to the routing in IPv4. Due to more levels of hierarchical addresses, less amount of memory is required to store routing tables.

In IPv6, only source nodes can perform packet fragmentation; intermediate routers on the packet's delivery path cannot [3]. Intermediate routers drop the packets that are larger than the network Maximum Transfer Unit (MTU) and send an ICMP error message to the source. On the other hand, in IPv4, not only source nodes can perform packet fragmentation, but also intermediate routers along a packet's delivery path. As a result, IPv4 slows down the routing process.

The term ICMPv6 stands for "Internet Control Message Protocols for Internet Protocol Version 6". ICMPv6 is described in RFC 2463. With numerous changes, it is made to replace "Internet Control Message Protocols for Internet Protocol Version 4" (ICMPv4). Like ICMPv4, ICMPv6 is fully implemented by IPv6.

The term MLD stands for "Multicast Listener Discovery". It derives from version 2 of IPv4's Internet Group Management Protocol (IGMPv2) and is described in RFC 2710. The MLD message types are actually subset of the set of ICMPv6 message types and not IGMP message types.

RIPng is a routing protocol used by routers to compute routes through an IPv6-based network. It is described in RFC 2080.

The corresponding standards of these protocols are described in nature language in RFCs. In order to avoid ambiguous interpretation and incorrect implementation of standards, it is necessary to formalize the protocols and facilitate their verification, validation and test suite generation. Exhaustive testing is not practical that is why only selected test cases are applied. The question of these test cases' quality is fundamental. The formal methods are used to assist for protocol simulation, validation and automatic test suite generation.

This thesis presents formal specifications in SDL [12] and test cases in TTCN for four main IPv6 protocols. The formalization of these protocols is accomplished in two steps. The first step includes the extraction and modeling of related RFCs. One of the encountered difficulties in this step is to extract the correct information from RFCs. In order to extract the information correctly, it is necessary to define the functional behaviours of the protocol by applying semiformal methods such as use case and sequence diagram techniques. Frequently, the ambiguous or lack of description of the parts in the standards causes difficulties. This step relates to the abstract level of protocols.

The second step consists of writing the SDL code. The Telelogic ObjectGEODE toolset [6] is used for specifying and simulating the specification. The SDL model is composed in SDL Editor and is simulated with the SDL simulator. The simulator acts as an SDL debugger. It also automatically detects design errors, such as deadlock and answer unexpected in a decision, etc. The simulating of the model performs partial verification and validation of the model.

A test case is generated from a test purpose and an SDL specification. The TestComposer is a test generator. It takes an SDL specification and test environment specification to compute a test purpose using reachability analysis technique and to generate the corresponding test case. The test case can be saved in TTCN produce module.

The remaining of the thesis is organized as follows: chapter 2 introduces and comments related work. Chapter 3 briefly introduces main IPv6 protocols. Chapter 4 introduces SDL with its supporting tool – ObjectGEODE toolset. While chapter 5 describes the applied approaches and corresponding examples. Finally, chapter 6 concludes the thesis and presents future work. Complete test cases of the protocols will follow the latter.

2. Related Work

IPv6 cannot be successfully deployed unless equipment manufactures and service providers can prove its conformance to the standards and interoperability with IPv4. A complete solution is to offer the full range of IPv6 testing.

IPv6 protocols are defined by more than 60 RFCs. These RFCs are described in natural language; thus, they are ambiguous and incomplete. Moreover, there is a high probability of misunderstanding and misinterpreting during implementation of such complex and large documents such as IPv6 RFCs. Furthermore, the implementation of IPv6 RFCs are inconsistent since they are different among developers. Due to misunderstanding, misinterpretation and different implantations and incompleteness of RFCs, the network products and services are possibly incompatible and inconsistent. Therefore, conformance testing is used to identify these problems earlier in development stage.

Conformance testing is the process of verifying if an implementation is consistent with a particular standard, specification, or environment. It is exclusively concerned with external behaviours of an implementation. Conformance testing is not intended to be exhaustive, and a successfully passed test suite does not mean that the implementation is 100-percent perfect. However, it does ensure, with a certain confidence, that the implementation is consistent with its specification, and it does increase the probability of an implementation interworking.

With the deployment of IPv6 protocols, few projects have been built to perform conformance testing and interoperability testing of these protocols. Among those few, there are TAHI project, MSR IPv6 Verification project and ARMOR project.

TAHI project [1] is the joined effort of the University of Tokyo and the Yokogawa Electric Corp. Its objective is to develop and provide the verification technology for IPv6. It provides a test package, which consists of conformance testing tool and interoperability testing tool. These testing tools require FreeBSD operating system, more than one Ethernet interfaces and Perl Library. FreeBSD system offers advanced networking, performance, security and compatibility features. It supports IPv6 Protocols. Perl API is used for creating a packet, receiving a packet and packet memory control. Principally, test will be performed by throwing a packet and checking the response. The test results are output to a HTML file (index.html).

The advantage of TAHI project is using free system FreeBSD, which provides many advanced features and powerful Internet solutions. The disadvantage of this project is that the specifications of requirements are written in Perl script, which is an informal language and the test result is reported in a HTML file, which is not a formal notation.

MSR IPv6 (Microsoft Research IPv6) Verification project [2] was built by the Institute for System Programming of Russia Academy of Sciences (ISPRAS). It aims at showing the applicability of formal techniques in specification and verification of complex API like IPv6 implementation. CTesK toolkit is used for test development automation. In CTesK, SeC (Specification Extension of C) language is used for test development. One of the test developments is the formal specification of the requirements of system under test. SeC translator translates SeC to C, which can be compiled into an executable module. Report Generator gets information about test execution. The information is reported in a HTML file.

The advantage of MSR IPv6 Verification project lay in using most powerful tool for unit and integration testing. The tests can be developed based on specification only, therefore, the tests are independent from implementation. The disadvantage of this project is that the test development is described by SeC language, which is not a formal language. The test result is reported in a HTML file, which is not a formal notation.

Moreover, both of TAHI project and MSR IPv6 Verification project do not have formal method to verify and validate their specification from requirements.

In the current thesis, SDL is used to write the formal specifications of IPv6 protocols. Powerful commercial toolset ObjectGeode, which supports SDL formal language, is used for the verification and the validation of the formal specification. TestComposer is a test generator in ObjectGeode. It is used to automatically generate abstract test suites from the formal specification and selective test purposes. The generated test suites are stored in a standard notation TTCN.

Now that the related work are summarized and commented, we will give an overview of the protocols used in this thesis.

3. Overview of the Main IPv6 Protocols

This chapter gives a brief overview of the main IPv6 protocols: IPv6, ICMPv6, MLD and RIPng. It begins with the basic concepts of the Open System Interconnection (OSI) model and IP stack. Then, the requirements of the main IPv6 protocols are described briefly.

A protocol is a set of rules used to enforce a standardized, structured language for the communication between multiple parties. Using the same protocols, two different systems can communicate regardless of their underlying architectures. It is easy to define protocols for two systems. However, it is very difficult to define protocols for any number of devices. The International Standardization Organization (ISO) has overcome the problem of communication between many devices at once by developing the OSI model. This model is a seven-layers model. Each of these seven layers performs a small, well-defined set of tasks and communicates with the layers directly above and below it.

A protocol stack is a set of a certain number of such layers and is organized in such a way that the highest level of abstraction lives at the top layer. Each layer of a stack builds its function upon the services provided by the layer immediately below it. It adds a header containing layer-specific information to the data packet. Using this concept, a source node encapsulates data and sends the data to the destination node. The destination node extracts layer information and process data accordingly.

In OSI model, the highest layer is called application layer: it works directly with the user or application program. The presentation layer is responsible for presenting data in a format its user can understand. The session layer allows applications on two different computers: whether it is to establish a session, or a logical connection. The forth layer is called transport layer. It is the lowest layer that handles primarily with end-to-end communication. The transport layer may determine which network is to be used for communication. The network layer deals with routing strategies; it is the highest layer in subnet. It handles routing information and maintains customer billing. The network layer provides the transport layer with the ability to establish end-to-end communication. This ability allows the transport layer to do its tasks without worrying about the detailed transmission between two stations. Data link layer supervises the flow of information between adjacent network nodes. Finally, the physical layer (the lowest layer in OSI model) transmits data bits over a network.

Internet is a collection of networks. Internet protocol (IP) is the standard that defines the manner for interconnecting end systems across multiple networks in the network layers. IP is implemented in each end system and in each router that provides connection between networks. A source end system (called source node) encapsulates higher-level data in an IP protocol data unit (called IP packet) for transmission. This IP packet is passed through one or more networks and connecting routers, to reach its final destination system (called destination node). A router is an essential device in the Internet; it allows a packet to pass through different networks.

As the Internet has grown, it shows the limitation of the current version IPv4. Therefore, the Internet Engineering Task Force issues the demand for proposals for next-generation IP in July of 1992. In response to this demand, 'The Recommendation for the IP Next Generation Protocol," is issued in January 1995. Finally, the IPv6 standard is issued in December 1998.

Figure 3.1 is graphic overview of IP stack. It contains application layer, transport layer, network layer, link layer and physical layer. Application layer protocols support network applications. Such as file transfer protocol, small mail transfer protocol, and hypertext transfer protocol. Transport layer protocols perform host-to-host data transfer. Such as TCP and UDP. Network layer protocols route data from source to destination. They allow data to be transferred over multiple networks. Internet Protocol and routing protocols are this layer protocols. Link layer protocols allow data to be transferred between neighboring network element of LAN (Local Area Network). Such as PPP and Ethernet. Physical layer transfers data in bits.

In the IPv6-based network, all nodes must implement IPv6, ICMPv6, ND and MLD. Only routers implement routing protocols. To simplify the implementation, RIPng protocol is selected as routing protocol. Due to time limitation, the core IPv6 protocols IPv6, ICMPv6, MLD and RIPng protocols are modeled.

IPv6 protocol defines the packet format and some possible options and validation rules. If an IPv6 node failed one or more validation constraints, an ICMPv6 error message might be sent to the packet's sender. RIPng defines the packet format and rules to allow routers to exchange information for computing routes over an IPv6 network. MLD defines the packet's format and behaviors to allow routers to discover the multicast listeners on its directly attached link.

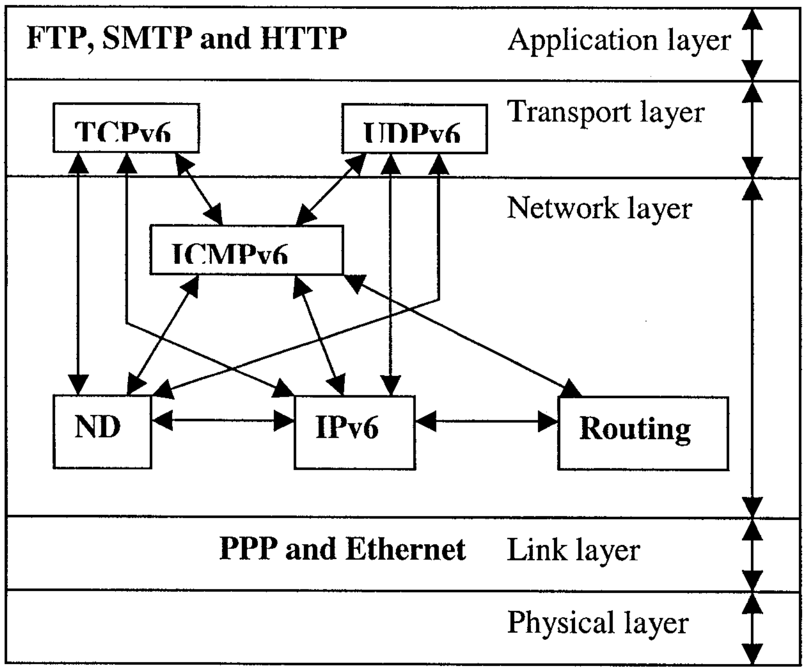


Figure 3.1 Overview of main IPv6 protocols

The rest of this chapter is organized as follows: in next section, the requirements of IPv6 protocol are described. In section 3.2, the brief overview of ICMPv6 protocol is given. In section 3.3, the basic requirements of MLD protocol are given. The last section describes the fundamental specification of RIPng protocol.

3.1 IPv6

IPv6 [3] is the new version of Internet Protocol. It was designed to overcome the inadequacies and limitation of IPv4. And indeed IPv6 did expand addressing capabilities by increasing IP address from 32 bits to 128 bits, thus it can support more levels of addressing hierarchy, a much greater number of addressable nodes, and simpler auto-configuration of addresses. Moreover, a "scope" field is added to multicast addresses so as to improve the scalability of multicast routing. In addition, a new type of address called "anycast address" is defined, and used to send a packet to any one of a group of nodes. Furthermore, IPv6 simplified header format by dropping some headers or making some headers optional. Hence it reduces the common-case processing cost of packet handling and limits the bandwidth cost of the header.

3.1.1 Packet Format

An IPv6 packet has a general format that includes IPv6 header, extension headers, and transport layer data. Extension headers are optional and placed between IPv6 header and transport layer data. A packet can have extension headers from zero to maximum seven. These extension headers are 1) Hop-By-hop option header, which defines special options that are required by each node along the way to the final destination; 2) routing header, which provides one or more router to be passed on the way to a packet's destination; 3) fragment header, which contains fragmentation information for reassembling; 4) authentication header, which provides authentication information; 5) encapsulation security payload header, which insures privacy; 6) destination option header1, which contains optional information that will be examined by the destination router; 7) destination option header2, which contains optional information that will be examined by the destination host. Each one of those extension headers is identified by a distinct next header value.

3.1.1.1 IPv6 Header Format

An IPv6 header has a fixed length of 40 octets. It includes eight fields, which are: Internet Protocol version field, Traffic Class field, Flow Label field, Payload Length field, Next Header field, Hop Limit field, Source Address field and Destination Address field. The value of the Internet Protocol version field is six, which means the Internet protocol's version is six. The traffic class is available for use by originating nodes or forwarding routers to identify and distinguish different classes or priorities of IPv6 packets. The Flow Label may be used by a source to label sequences of packets for which it request special handing by the IPv6 routers. The Payload Length is the length of the payload, which is the rest of IPv6 packet. The Next Header identifies the type of header immediately following the IPv6 header. The Hop Limit is decremented by 1 by each node that forwards the packet. The Source Address is a 128-bit address of originator of the packet. The Destination Address is a 128-bit address of the intended recipient of the packet.

3.1.1.2 Options Format

It is an option definition that includes Option Type, Opt Data Len and Option data. The Option Type identifies the type of option. It may not be recognized during the IPv6 node processing. In such case, the action must be taken according to the highest-order two bits of the Option Type. The Opt Data Len specifies the length of the option data. The Option Data is a variable-length data of option.

3.1.1.3 Hop-by-Hop Options Header Format

The hop-by-Hop options header is identified by a Next Header value of 0 in the IPv6 header. If it is present, it must be examined by each node along the way the packet passes. It contains next header, which identifies the type of header immediately following the Hop-by-Hop option header, and HdrExtLen, which is the length of the hop-by-hop Options header, and options.

3.1.1.4 Routing header format

The Routing header is identified by a next header value of 43 in the immediately preceding header. If it is present, it must be examined by all destination nodes. At that point, the IPv6 packet and routing header contents are updated and the packet is forwarded. Routing header contains Next Header, Hdr Ext Len, Routing Type, Segment Left, and type-specific data. The Next Header identifies the type of header immediately following; the Hdr Ext Len is the length of the routing header; Routing Type identifies a particular routing header; the Segment Left indicates the number of intermediate nodes left to be visited. The type-specific data is a variable-length field and the routing type determines its format.

3.1.1.5 Fragment header format

The fragment header is identified by a Next Header value of 44 in the immediately preceding header. If it is present, it can only be processed by final destination node. A fragment header contains Next Header field, Reserved field, Fragment Offset field, M flag field and Identification field.

The Next Header identifies the initial header type of the fragmentable part of the original packet; the Fragment Offset is the offset of the data following this header and relative to the start of the fragmentable part of the original packet; the M flag indicates whether the packet is the last fragment; the Identification identifies the fragments.

3.1.1.6 Destination Options header format

The Destination options header is identified by a Next Header value of 60 in the immediately preceding header; its format is the same as the hop-by-Hop options header.

3.1.2 Packet Size Issues

IPv6 requires that every link in the Internet have a Maximum Transmission Unit (MTU) of 1280 octets or greater. On any link that cannot convey a 1280-octet packet in one piece, link-specific fragmentation and reassembly must be provided at a layer below IPv6. It is recommended for links to be configured with MTU of 1500 octets or greater to avoid incurring IPv6-layer fragmentation.

This section has a brief overview of IPv6 protocol. The detailed information about this protocol is described in RFC 2460. The next section will give a brief overview of ICMPv6 protocol.

3.2 ICMPv6

ICMPv6 [4] is defined for IPv6. It is an integral part of IPv6 and MUST be fully implemented by each IPv6 node. The header is identified by a next header value of 58 in the immediately preceding header. ICMPv6 affords message packets to report errors encountered during packets processing and to perform other special tasks. The messages are classified into two categories: error messages and informational messages. An ICMP message is generated only once no matter if it can or cannot be delivered in order to avoid an endless transmission of ICMP messages.

3.2.1 Error Messages Format

An error message is identified by its message type. The value of the message type is marked on a scale of 0 to 127.

3.2.1.1 Destination Unreachable Message

A router generates a destination unreachable message when it is unable to send the packet to its destination. The type value of the message is 1. The code value inside the message defines four reasons for delivery failures: 1) no route to destination (code=0); 2) communication with destination administratively prohibited (code=1); 3) unassigned address (code=2); 4) unreachable address (code =3); 5) unreachable port (code=4). The graphic view is figure 3.2.1

Type	Code	Checksum
Unused		
As much of invoking packet as will fit without the ICMPv6 packet exceeding the minimum IPV6 MTU		

Figure 3.2.1: destination unreachable message format

3.2.1.2 Packet Too Big Message

A packet too big message is generated when the packet cannot be forwarded because its size is larger than the MTU of the next-hop link. The type value of the message is 2. The code value is 0. The graphic view is figure 3.2.2.

Type	Code	Checksum
MTU		
As much of invoking packet as will fit without the ICMPv6 packet exceeding the minimum IPV6 MTU		

Figure 3.2.2 packet too big message format

3.2.1.3 Time Exceeded Message

A time-exceeded message is generated when a router receives a packet with zero Hop Limit, or the router decrements the packet's Hop Limit to zero, or reassembles fragment packets timeout. The type value of the message is 3. The code value is 0 if the problem is Hop limit exceeded; is 1 if the problem is fragment reassembly time exceeded. The graphic view is fig. 3.2.3.

Type	Code	Checksum
Unused		
As much of invoking packet as will fit without the ICMPv6 packet exceeding the minimum IPV6 MTU		

Figure 3.2.3 time exceeded message format

3.2.1.4 Parameter Problem Message

A parameter problem message is generated when IPv6 nodes find a problem in a field of the IPv6 header, or extension headers. The message type is 4. The code value is 0 if erroneous header field is encountered. It is 1 if unrecognized next header type is encountered. It is 2 if unrecognized IPV6 option is encountered. The pointer field in the message identifies the octet offset within the invoking packet where the error was detected. The graphic view is figure 3.2.4.

Type	Code	Checksum
Pointer		
As much of invoking packet as will fit without the ICMPv6 packet exceeding the minimum IPV6 MTU		

Figure 3.2.4 parameter problem message format

3.2.2 Informational Messages

An information message contains message type, code, identifier and sequence number. The value of the message type goes from 128 to 255. It identifies an information message.

3.2.2.1 Echo Request Message

The type value of an Echo request Message is 128. The code value is 0. The identifier and sequence number aid in matching Echo Replies to this Echo Request. The graphic view is figure 3.2.5.

Type	Code	Checksum
Identifier		Sequence Number
Data …		

Figure 3.2.5 Echo Request message format

3.2.2.2 Echo Reply Message

An Echo Reply message is sent to respond to an Echo Request. The type value is 129, and the code value is zero. The identifier and sequence number are identical to those values in the Echo Request. The graphic view is figure 3.2.6.

Type	Code	Checksum
Identifier		Sequence Number
Data ...		

Figure 3.2.6 Echo Reply message format

This section has a brief overview of ICMPv6 protocol. The detailed information about this protocol is described in RFC 2463. The next section will give a brief overview of MLD protocol.

3.3 MLD

MLD [13] is a sub-protocol of ICMPv6. It is used by an IPv6 router to discover if there are multicast listeners on its directly attached links and to determine specifically which multicast addresses are of interest to those nodes. Then routers use this information to ensure that multicast packets are delivered to the interested nodes.

MLD is an asymmetric protocol. It specifies different behaviours for multicast listeners and for routers. When a router is a multicast listener to itself for some multicast addresses, the router performs both parts of the protocol.

3.3.1 Message Format

MLD message types are a subset of the set of ICMPv6 message types, and MLD messages are identified in IPv6 packets by preceding next header value of 58. They are sent with a link-local IPv6 Source Address, an IPv6 Hop Limit of 1, and an IPv6 Router Alert option in a Hop-by-Hop Options header. Figure 3.3.1 is the format of the MLD messages:

Type (8 bits)	Code (8 bits)	Checksum (16 bits)
Maximum Response Delay (16 bits)	Reserved (16 bits)	
Multicast Address (128 bits)		

Figure 3.3.1 MLD message format

Where the 'Type' specifies the MLD message. The detailed descriptions are: the type of Multicast Listener Query is 130, the type of Multicast Listener report is 131, and the type of Multicast Listener Done is 132.

In figure3.3.1, the 'Code' is initialized to 0 by the sender, and is ignored by receivers. It is used for further qualifying the MLD message. The 'Maximum Response

Delay' is set to 0 by the sender and is ignored by receivers in all messages except in Query message. In Query messages, the 'Maximum Response Delay' specifies the maximum allowed delay before sending a responding Report. The 'Reserved' field is initialized to zero by the sender and is ignored by receivers. The 'Multicast Address' has different meaning in different messages. The detailed descriptions are shown in figure3.3.2:

Message Type	Description of Multicast Address
In General Query	Is set to 0
In Multicast-Address-Specific Query	Is set to a specific IPv6 multicast address
In a Report	Is set to a specific multicast address where the message is listening
In Done message	Is set to a specific multicast address where the message is ceasing to listen

Figure 3.3.2 description of multicast address

3.3.2 Protocol Description

Routers use MLD to discover which multicast addresses have listeners on each of their attached links. For each attached link, a router selects one of its link-local unicast addresses of that link to be used as the IPv6 source address in all MLD packets it transmits on that link. For each interface over which the router is operating the MLD protocol, the router must configure that interface to listen to all link-layer multicast address that can be generated by IPv6 multicasts.

With respect to each of its attached links, a router may act either as a Querier or a Non-Querier. Normally, there is only one Querier per link. All routers start up as a Querier on each of their attached links. At the beginning, there is one Querier for each link. It sends a General Query on that link and starts initial General Query Timer. After the General Query Timer expired, the Querier sends another General Query and starts

General Query Timer again. The Querier repeats this action until it heads a Query Message whose IPv6 Source address is numerically less than its own selected address for that link. At that moment the Querier must start Other Querier Present Timer, and then become a Non-Querier. A Non-Querier will become a Querier if Other Present Querier Timer expired without receiving any Query from a router with an address less than its own.

When a router receives a report from a link, it will update the router's list of multicast address having listeners on that link accordingly. When a router receives a Done message from a link, if it is in Non-Querier state, it must ignore the Done message. Otherwise, the router checks whether the multicast address of the link is in the multicast address list. If the address is in the list, the router sends "Last Listener Query Count" Multicast-Address-Specific Queries, one every "Last Listener Query interval" to that multicast address. If no reports are received for the address from this link after "maximum response delay", the router assumes that there is no listener on the address of the link. The figure 3.3.3 describes the state diagram of the router.

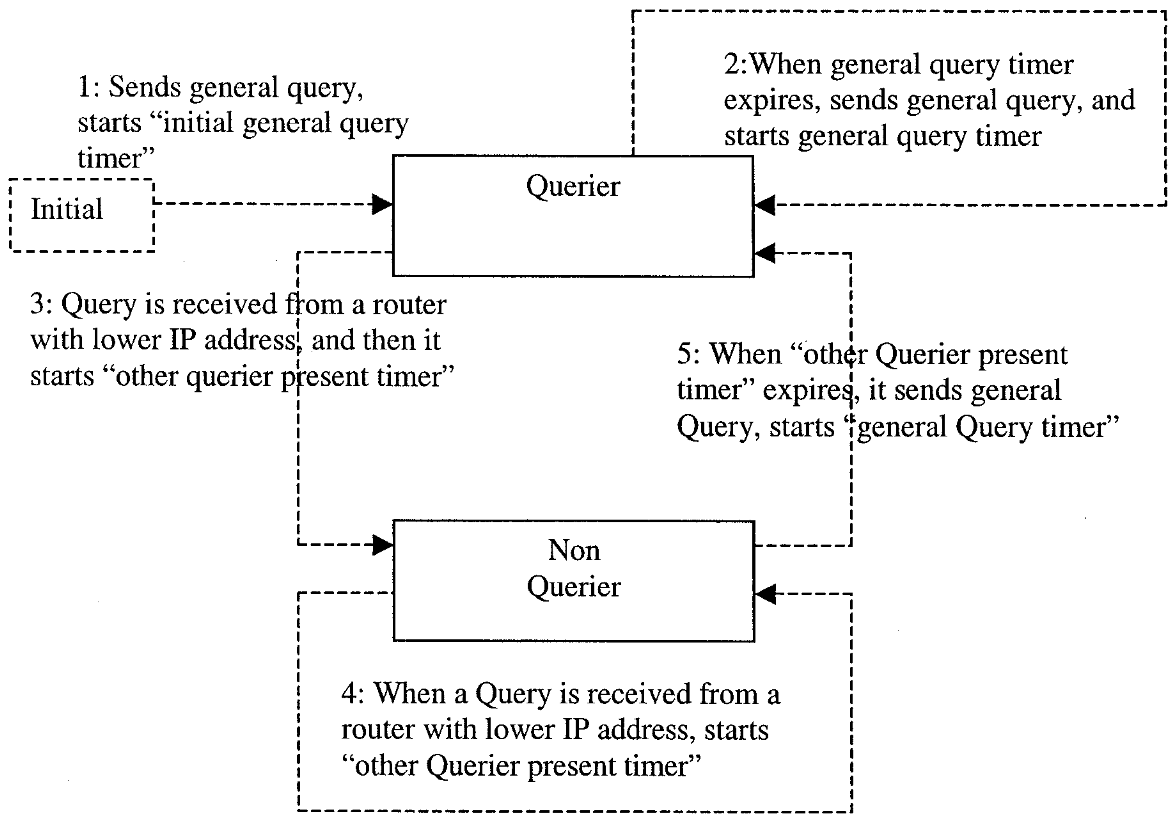

1: Sends general query, starts "initial general query timer"
2:When general query timer expires, sends general query, and starts general query timer
Initial
Querier
3: Query is received from a router with lower IP address, and then it starts "other querier present timer"
5: When "other Querier present timer" expires, it sends general Query, starts "general Query timer"
Non Querier
4: When a Query is received from a router with lower IP address, starts "other Querier present timer"

Figure 3.3.3 router states diagram

With respect to any single IPv6 multicast address on any single interface, a node may be on Non-listener state, Delaying-Listener state, or Idle Listener state. The Non-listener is the initial state of a node. At this state, the node is not listening to the address on the interface. When a node wants to receive a multicast packet, it sends a Report to Querier router attached to its link. And then the node is on Delaying-Listener state.

On the Delaying-Listener state, a node performs four actions up to its receiving and timer. If a Query is received, it sets delay timer for the multicast address to which it is listening on the interface from which the Query was received, excluding the Link-scope all nodes address and any multicast addresses of scope reserved or node-local. If a Report is received, the node stops the timer, clear the flag and then changes to Idle-

Listener state. If the timer expired, the node sends a report, sets the Flag and then changes to Idle-Listener state. On the Idle-Listener state, a node starts timer and changes to Delaying-Listener state if it receives a Query.

A node could stop listening by sending a 'Done' message to Querier Router when it is on Delaying-Listener state or Idle-Listener state. The figure 3.3.4 is a node state diagram with the corresponding behaviours:

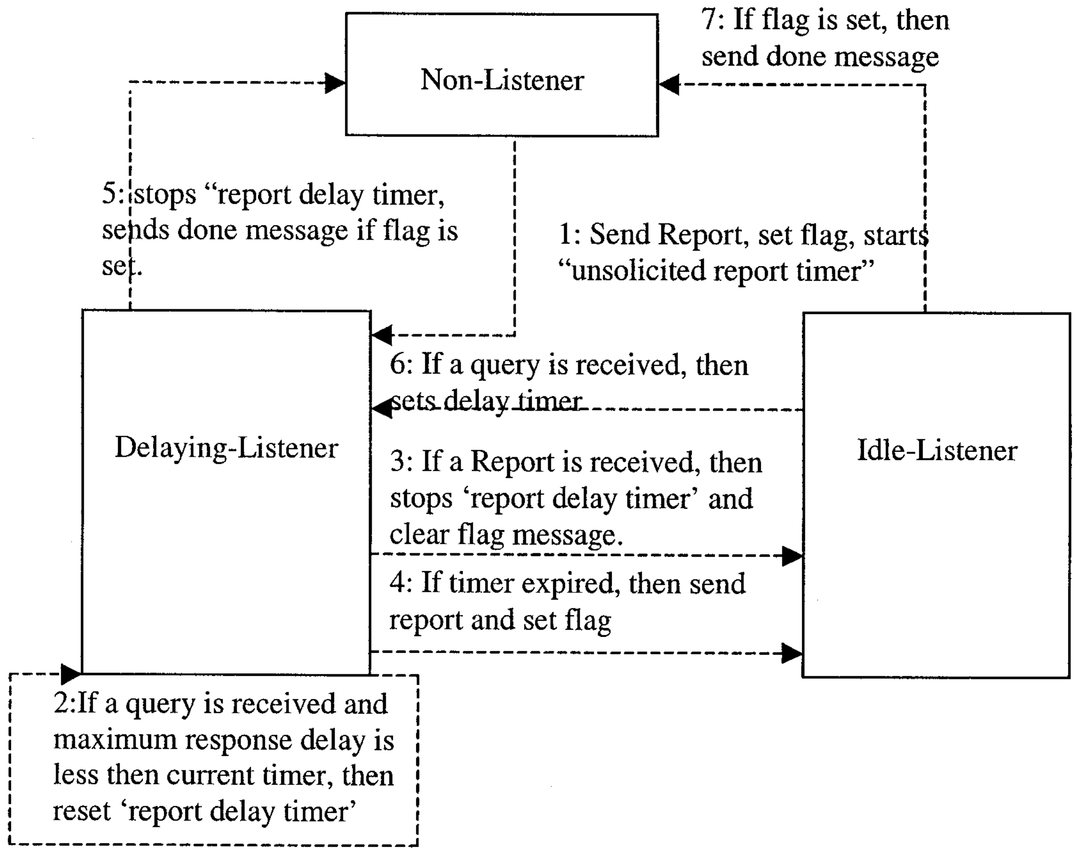

Non-Listener
7: If flag is set, then send done message
5: stops "report delay timer, sends done message if flag is set.
1: Send Report, set flag, starts "unsolicited report timer"
Delaying-Listener
Idle-Listener
6: If a query is received, then sets delay timer
3: If a Report is received, then stops 'report delay timer' and clear flag message.
4: If timer expired, then send report and set flag
2:If a query is received and maximum response delay is less then current timer, then reset 'report delay timer'

Figure 3.3.4 node states diagram

This section has a brief overview of MLD protocol. The detailed information about this protocol is described in RFC 2710. The next section will give a brief overview of RIPng protocol.

3.4 RIPng

The term RIPng [5] stands for Routing Information Protocol New Generation for an IPv6 Internet. It was designed to work as an Interior Gateway Protocol (IGP) in networks of moderate-size. It is not intended to be used in a more complex environment.

RIPng is based on Distance Vector Algorithm to allow routers to exchange routing information through an IPv6 based network. It should be implemented only in routers. Any router that uses RIPng is assumed to have interfaces to one or more networks. The protocol relies on access to certain information about each of these networks, the most important of which is its metric. The metric is an integer between 1 and 15, inclusively. Moreover, each network will have an IPv6 destination address prefix and prefix length associated with it.

The protocol is limited to networks whose longest path is 15 hops. It depends upon 'counting to infinity" to resolve certain unusual situations, therefore the resolution of routing loop may require either much time or bandwidth. Furthermore, the protocol is not appropriate for the situation where the routes need to be chosen on real-time parameters.

3.4.1 Message Format

RIPng is an UDP-based protocol. Each router that uses RIPng has a routing process that sends and receives datagrams on UDP port number 521. The RIPng packet format is in figure 3.4.1:

Command (1 octet)	Version (1 octet)	Must be zero (2 octets)
Route Table Entry 1 (20 octets)		
Route Table Entry N (20 octets)		

Figure 3.4.1 RIPng packet format

In figure 3.4.1, the 'command' is used to specify the purpose of this message; the 'version' contains the version number of the protocol. The commands implemented in version 1 are Request and Response. A Request Message is a request for a system to send all or part of its routing table. A Response Message contains all or part of the sender's routing table. It may be sent in response to a request, or be an unsolicited routing update generated by the sender. Each of these messages contains a list of RTEs. Each Route Table Entry (RTE) has the following format:

IPv6 prefix (16 octets)		
Route tag (2 octets)	Prefix Len (1 octet)	Metric (1 octet)

Figure 3.4.2 routing table entry

In figure 3.4.2, 'IPv6 prefix' is prefix of the destination that is stored in 16 octets in network byte order. The route tag field is an attribute assigned to a route that must be preserved and readvertised with a route. The intended use of the route tag is to provide a method of separating 'internal' RIPng routes from "external" RIPng routes, which may have been imported from an Exterior Gateway Protocol or another Interior Gateway Protocol.

The 'Prefix Len', prefix length field, is the length in bits of the significant part of the prefix (a value between 0 and 128 inclusive) starting from the left of the prefix.

The 'Metric' contains a value between 1 and 15 inclusively, specifying the current metric for the destination; or the value 16 (infinity), which indicates that the destination is not reachable.

RIPng provides the ability to specify the immediate next hop IPv6 address a packet must use to reach the destination specified by a route table entry (RTE). A next hop RTE is identified by a value of 0xFF in the metric field of an RTE. The prefix field specifies the IPv6 address of the next hop. The route tag and prefix length in the next hop RTE must be set to zero by the sender and ignored by the receiver. The purpose of the next hop RTE is to eliminate packets being routed through extra hops in the system.

3.4.2 Addressing Considerations

The distinction between network, subnet and a host routes does not need to be made for RIPng because an IPv6 address prefix is unambiguous. Any prefix with a prefix length of zero is used to designate a default route. A default route is used when it is not convenient to list every possible network in the RIPng updates, and when one or more routers in the system are prepared to handle traffic to the networks that are not explicitly listed. These "default routers" use the default route as a path for all datagrams for which they have no explicit route.

3.4.3 Timers

Every 30 seconds, the RIPng process is awakened to send an unsolicited response message to every neighbouring router. If there are many routers on a single network, the unnecessary collision will happen on broadcast network. In order to avoid these unnecessary collisions, the implementations are required to take one of these two protections: 1) The 30-second updates are triggered by a clock whose rate is not affected by system load or the time required servicing the previous update timer. 2) The 30-

second timer is offset by a small random time (+/- 0 to 15 seconds) each time it is set. The offset derives from: 0.5 * the update period.

There are two timers associated with each route, a "timeout" and a "garbage-collection time." Upon expiration of the timeout, the route is no longer valid; however, it is retained in the routing table for a short time so that neighbours can be notified that the route has been dropped. Upon expiration of the garbage-collection timer, the route is finally removed from the routing table.

3.4.4 Input Processing

This section describes the handling of datagrams received on the RIPng port. The processing will depend on the value in the command field.

3.4.4.1 Request Message

A Request is used to ask for a response containing all or part of a router's routing table. Usually, a Request is sent as multicast from the RIPng port. Nevertheless, if the routing table of only a single router is needed, the Request should be sent directly to that router from a UDP port other than the RIPng port.

The Request is processed entry by entry. For each entry, the router looks up the destination in its routing database; if there is a route, the router puts that route's metric in the metric field of the RT; If there is no explicit route to the specified destination, the router puts infinity in the metric field. Once all the entries have been filled in, the router changes the command from Request to Response and sends the datagram back to the requestor.

However, some special cases have to be considered: if there are no entries, no response is given; if there is exactly one entry in the request, and if it is a next hop Route Table Entry, then this is a request for sending the entire routing table.

3.4.4.2 Response Message

A Response can be received in response to a specific query, an unsolicited response or a triggered update caused by a route change. The processing is the same no matter why the Response was generated. In order to ensure to update the routing table correctly, the Response must be validated. The validation includes several sending port number valid, source address valid, destination prefix valid, prefix length valid, and metric valid. The Response must be ignored if it is not from the RIPng port. If any other check fails, the router ignores that entry and proceeds to next entry.

Once the entry has been validated, the router updates the metric by adding the cost of the network on which the message arrived. If the result is greater than infinity, use infinity. That is: $\boxed{\text{metric} = \text{MIN (metric + cost, infinity)}}$

If there is no such route, the router adds this route to the routing table, unless the metric is infinity.

3.4.5 Output Processing

This section describes the processing used to create response messages that contain all or part of the routing table. Input processing, regular routing update, or triggered updates may trigger this processing.

3.4.5.1 Triggered Updates

There are two reasons why triggered updates require special handling. The first one is that practice knows that triggered updates can cause excessive loads on networks with limited capacity or networks with many routers on them. Therefore, the protocol requires limiting the frequency of triggered updates. The second one is that triggered updates do not need to include the entire routing table. In principle, only those routes that have changed need to be included. Therefore, messages generated as part of a triggered update must include at least those routes that had their route change flag set. When a triggered update is processed, messages should be generated for every directly connected network. If no routes need be sent on that network, the update may be omitted.

3.4.5.2 Generating Response Message

This section describes how a Response message is generated for a particular directly connected network.

Normally, the IPv6 source address must be a link-local address of the possible addresses of the sending router's interface. It is important to use a link-local address because the source address is put into the routing table in the router that receives this Response. Therefore, If an incorrect source address is used, other routers may be unable to route datagrams.

However, the source address must be a globally valid address if replying to a unicast Request Message from a port other than the *RIPng* port.

4. SDL and Its Supporting Tool

This section is a brief overview of SDL and its supporting tool-ObjectGEODE toolset. It covers the basic concepts and main characteristics of SDL to answer the question: why and how to use SDL. It also covers essential functions of ObjectGEODE toolset to clarify the advantages of the toolset. All elements concerning SDL are described in section 4.1; all elements concerning ObjectGEODE toolset are described in section 4.2.

4.1 SDL

It is in 1972 that scientists began to develop SDL. In 1976, the first version made its appearance. Later in 1980, 1984, 1988, 1992, 1996,1999 and 2000, new versions followed. The first recommendation Z.100 [10] is issued by the International Telecommunications Union–Telecommunications Standardization Sector (ITU–T) in 1980. Each following version has better solution than previous version. The Object Oriented features were included in the language in 1992. This was extended in the latest version (SDL-2000) to give better support for object modeling and for code generation.

SDL [12] is a standard language for the description of a system. This language is designed for the specification of complex, event-driven, real-time, and interactive applications that involve many concurrent activities that communicate using discrete signals. It has a well-defined set of concepts and an unambiguous, clear, precise, and concise specification. It also has a precise basis for analyzing specification, determining consistency and implementing the conformance of a specification.

SDL is based on a set of extended finite state machines (EFSMs). These machines are independent and communicate asynchronously by using FIFO algorithm. A SDL system consists of structure, communication, behaviours, data, and inheritance components. The system structure and behaviour are described in section 4.1.1. Communication is described in section 4.1.2. In order to enrich explanations about communication in SDL, time and timer are described in section 4.1.3. Data and inheritance are described in section 4.1.4.

4.1.1 System Structure and Behavior

A SDL system contains four main hierarchical levels. The first level (top level of detail) is system, which can be partitioned into blocks and processes. The second level is blocks that can be divided into processes and procedures. The third level is processes that are described by extended finite state machines and procedures.

A system is an abstract machine that communicates with its environment [12]. The behaviour of a system is described by a number of processes. In SDL/GR model, a system is called a system diagram. Basically, a system diagram contains system's name, signal descriptions, channel descriptions, data type descriptions and blocks descriptions.

A block is a part of the system that performs conceptual functions as a unit. In SDL/GR model, a block is called a block diagram. Basically, a block diagram contains block's name, signal descriptions, signal route descriptions, channel-to-route connections, and process descriptions.

A process is an extended finite state machine. It can be created at system start. It can also be created by another process at run time. One or more instance of a process can exist simultaneously. Each instance has a unique process identifier (PId). Therefore, it is possible to send signals to individual instances of a process. The concept of processes and process instances that work autonomously and concurrently makes SDL a true real-time language.

4.1.2 Communication

SDL has two basic communication mechanisms: asynchronous signals and synchronous remote procedure calls. Both mechanisms can interchange and synchronize information between: 1) different processes, 2) different blocks, 3) SDL system and its environment. SDL defines channels to perform communication between different blocks of a system or communication between SDL system and its environment. It also defines signal route to perform communication between different processes within a block. Moreover, SDL defines time and timers to enrich the communication mechanism for real time applications.

4.1.3 Timer

Timer is an essential concept in a real time system. In SDL, Time and timers are defined in abstract manner. A timer must be described in a process. An inactivated timer can be activated and vice versa. An activated timer can expire within certain periods. When an SDL timer expires, the process that started the timer receives a notification (signal) in the same way as it receives any other signal. Actually, an expired timer is treated in exactly the same way as a signal.

4.1.4 Data and Inheritance Concept

SDL describes data type using basic predefined data type, such as integer, charstring, and Boolean. It also accepts abstract data type (ADT) and ASN.1. An ADT does not have specified data structure. However it specifies a set of values, a set of operators, and a set of equations. Hence ADT can hide data manipulation and algorithm. ASN.1 describes data using a set of predefined data types, and allows the definition of composed data types, such as groups of elements (called SEQUENCE), sequence of identical types (called SEQUENCE OF) and a type of alternatives (called CHOICE).

OO Inheritance concept of SDL is based on type declaration. Type declaration can be placed in different level. It is possible to structure and reuse information. Combining ADT and ASN.1, it is easy to define Protocol Data Unit in the SDL system. Therefore, SDL is well known and suited for telecommunication applications.

4.2 ObjectGEODE Toolset

ObjectGEODE is an environment that supports the SDL language. It can produce well-defined, complete and consistent specification, reduce the complexity of the specification, and generate test cases automatically.

ObjectGEODE toolset [6] provides graphical editors, an efficient simulator, an actual C/C++ code generator, and a test generation tool. Therefore, the designer can stipulate a system graphically, achieve a formal verification and validation of the system, create automatically the C or C++ code from the system, and generate test suites for conformance testing by using ObjectGEODE.

4.2.1 Graphic Editors

ObjectGEODE contains five graphic editors: UML Class Diagram Editor, MSC Editor, UML StateChart Editor, SDL Editor and SDL&MSC checker. The first four editors are used to create, modify, and view the diagrams of a system description: Architecture, Communication, State machines and Message Sequence diagrams. The latter ensures the consistency and compliance of a system description with notation rules. From a practical point of view, the designer only needs to use SDL editor combined with SDL&MSC checker to describe the architecture and communication successfully.

4.2.2 Efficient Simulator

The simulator provides debugging, verification and validation techniques to detect any modeling error. There are three modes: interactive mode, random mode and exhaustive mode. With interactive simulation (interactive mode), the user plays a role of environment by sending external signals or by increasing time. The system is observed as a white-box. In intensive simulation (random mode), the simulator automatically fires a series of transitions at random. In exhaustive simulation (exhaustive mode), the simulator exhaustively explores the model's behaviors under MSC and GOAL observers' monitoring.

4.2.3 Test Generation Tool

'TestComposer' is one of the test generation tools. It is contained in ObjectGeode toolset. The TestComposer [8] performs conformance testing by generating test suites for an Implementation Under Test (IUT). The tool takes SDL specification, test environment specification, and test purposes as input and generates a test case for each fed test purpose. The generated test cases are stored in the test case database. Finally, test suites are generated throughout the test cases database API, and are written a TTCN-mp file (Tree and Tabular Combined Notation-Machine Processing). The TTCN-mp file can be viewed in TTCN editor.

TTCN [9] is an ISO standard for describing test suites that are executed on software application. It has been developed for the description of test cases for OSI conformance test suites.

Now that the basic concepts have been established, we will pass to their applications.

5. SDL Modeling, Verification and Validation

In this chapter, an explanation of the methodology and techniques, which are used to formalize, verify and validate the model and generate test cases, is given. There are four fundamental steps [12]: 1) extracting technical information, 2) creating SDL model, 3) verifying and validating the SDL model, and 4) generating TTCN test cases. The first section treats the first of the four fundamental steps: 'extracting technical information.

5.1 Extracting Technical Information

This stage is an indispensable phase. During this phase, many ambiguities and incomplete information may be encountered, and any mistakes will lead to an erroneous implementation. It takes the highest cost to correct this kind of erroneous implementation during the software development life cycle. This stage is also the most challenging part of the implementation because of the necessity to fully understand the RFCs and to complete the ambiguities and incomplete information in the RFCs. The difficulty is even greater since RFCs make multiple self-references (one RFC often refers to other RFCs that refers to each other and to the first RFC). For example, in ICMPv6 (RFC 2463), the description of 'upper layer notification' is incomplete. It does not describe what should be notified when a node receives an error message. It does not describe the condition of passing Echo Request message to processes receiving ICMP message nor the condition of passing Echo Reply message to processes that did not originate the Echo Request message. The repeated use of "maybe" underlines the uncertainties lied forth by the RFCs. In IPv6 protocol (RFC 2460), it allows extension headers to occur in any order and any number of times in the same packet. However, it does not mention any solution for its implementation. There are many other examples, but we shall leave them unnoticed and stay in topic. Therefore, because of the high difficulty to understand and complete RFCs, it is helpful to present the extracted information and modeled it in any semi-formal ways before describing it with SDL formal language or proposing any implementation, especially considering the high cost of correcting mistakes made during this stage.

This thesis implements the protocols' mandatory parts that refer to "must' or "should" statements in RFCs. The technical information has been extracted carefully and correctly by using top-down design methodology. The extracted information is described in MSC semi-formally. In the rest of this chapter, IPv6 protocol is used as an example to demonstrate the general methodologies.

By learning IPv6 protocol, an IP node is a device that implements IPv6. A router is a node that forwards IPv6 packets, and a host is a node that runs application. Hosts of different networks can communicate through routers. In the rest of this thesis, the word router refers to the network layer of a router; the word host refers to the network layer of a host.

A router implements IPv6, ICMPv6, MLD and RIPng routing protocols. It may be an intermediate router or a destination router. Normally, an intermediate router forwards a packet to another router. A destination router plays the same role as an intermediate router, except that it not only examines hop-by-hop header, but also examines destination option header1 and routing header. At this moment, the routing header is updated by placing the next address to be visited in the IPv6 header and decrementing the segment left field in the routing header.

If a router receives a packet that contains errors, it may discard this packet, send an error message to the source, or forward this packet anyway.

A router in IPv6 does not fragment a packet. When it tries to forward a packet whose size is more than MTU, an error message will be sent to the source. These extracted information are described in MSC semi-formally as follows:

Scenario 1: Figure 5.1 shows that a router forwards a packet.

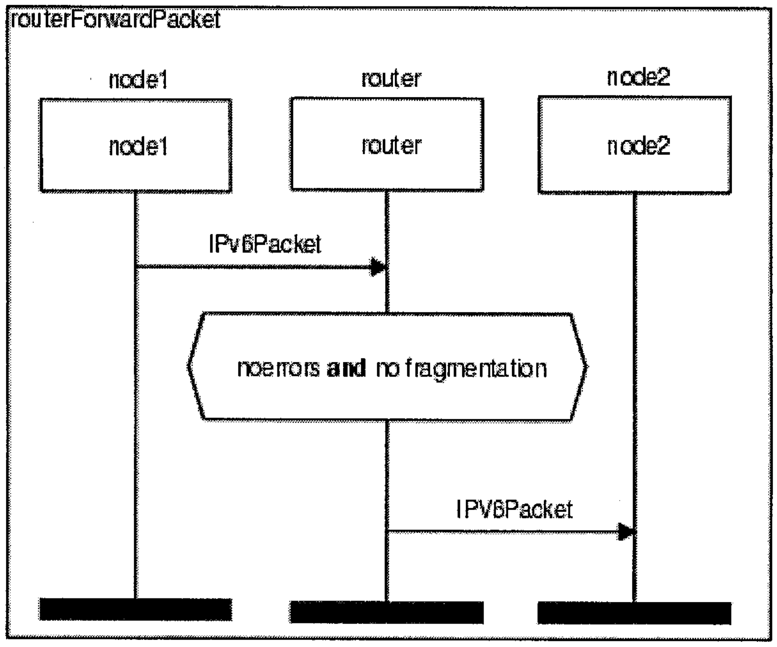


Figure 5.1 a router forwards a packet

Scenario 2: Figure 5.2 shows a router sends an error message.

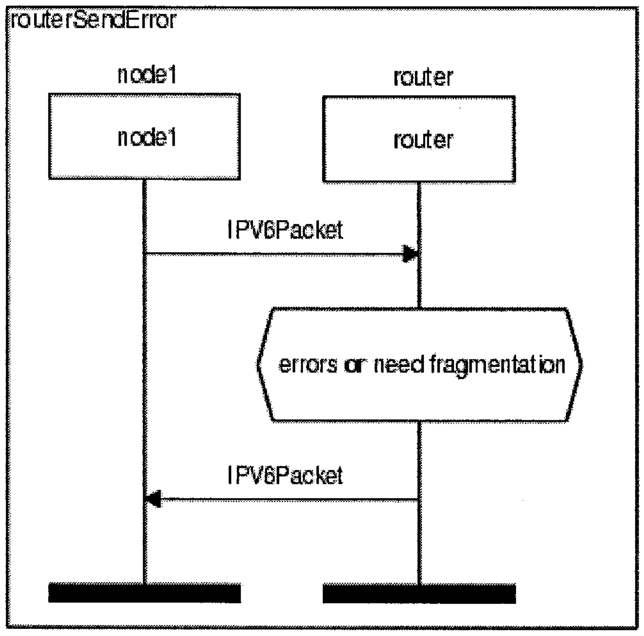


Figure 5.2 a router sends an error message

A host can send and receive a packet. If a host sends a packet whose size is more than MTU, then this host fragments this packet into several fragments, and sends these fragments to the destination. If a host receives fragments, it reassembles these fragments to the original packet.

A host has ability to examine a packet entirely. If a host is a source, it examines hop-by-hop header only (if the packet contains this header). Consequently, this host may send an error message to its upper layer, discard this packet, or send this packet to the destination. If a host is a destination, it examines all headers (whichever the packet contains). As a result, this host may send an error message to the source, discard this packet, or send PDU to its upper layer.

Scenario 3: Figure 5.3 shows that a host sends a packet.

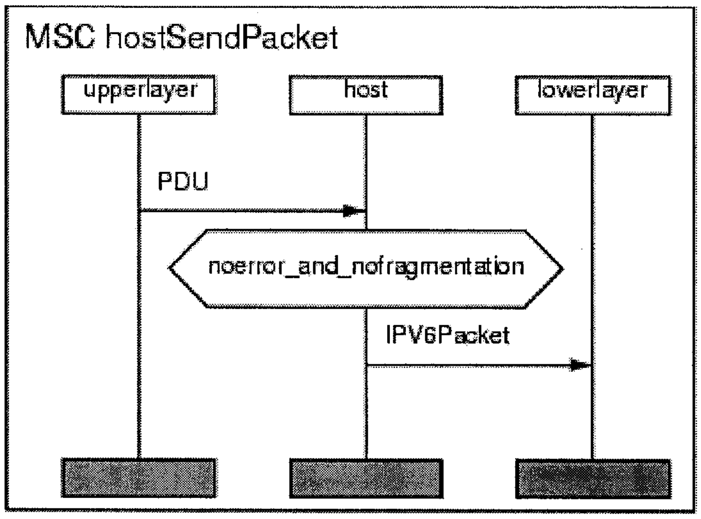


Figure 5.3 a host sends a packet

Scenario 4: Figure 5.4 shows a host sends fragments.

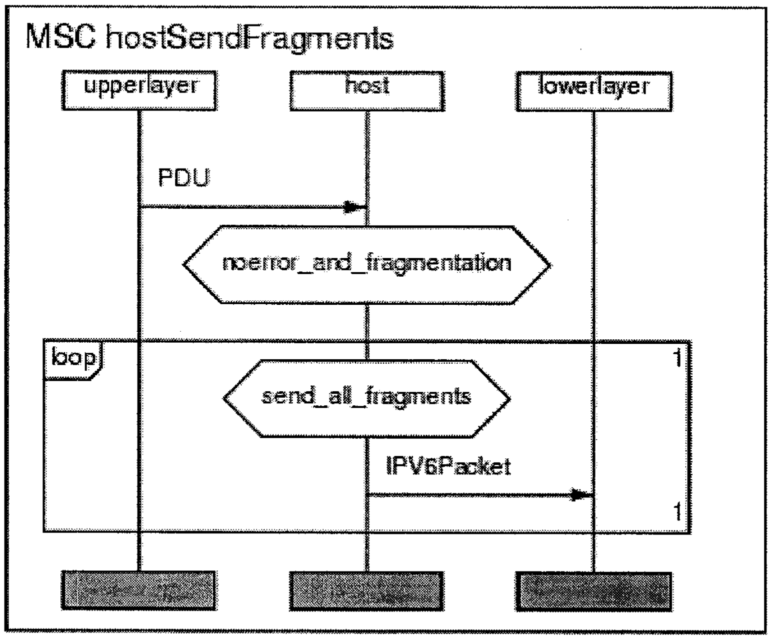


Figure 5.4 a host sends fragments

Scenario 5: Figure 5.5 shows a host sends PDU.

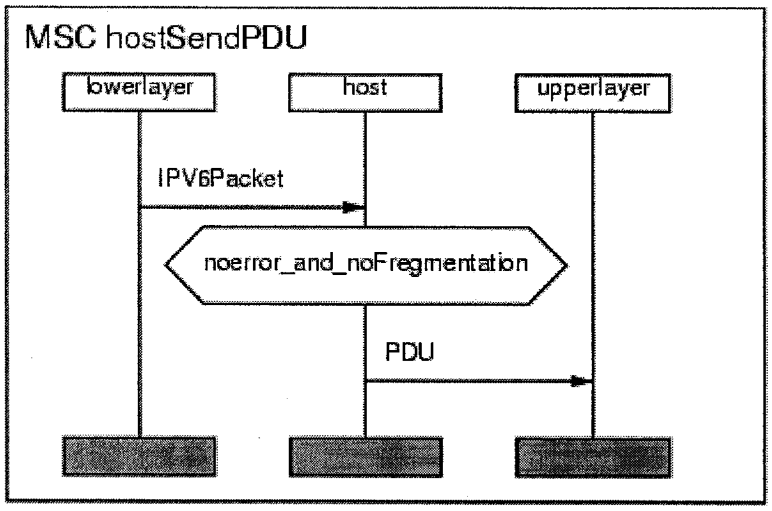


Figure 5.5 a host sends PDU

Scenario 6: Figure 5.6 shows a host reassembles fragments and sends PDU.

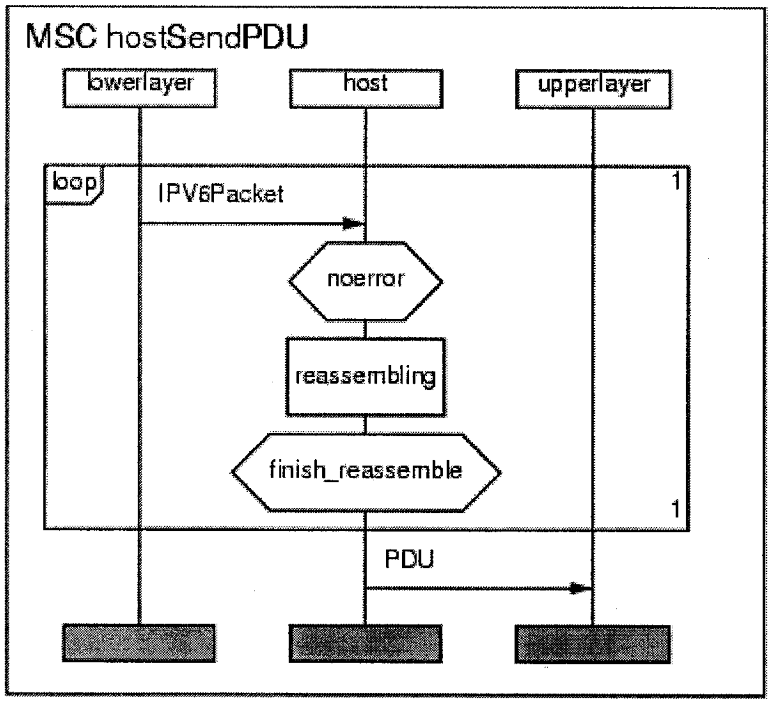


Figure 5.6 a host reassembles fragments and sends PDU

Scenario 7: Figure 5.7 shows a host sends an error message.

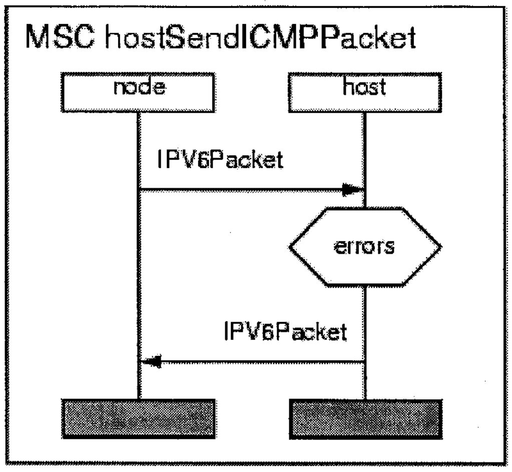


Figure 5.7 a host sends an error message

According to the scenarios presented above, we can describe a host or a router separately to accomplish the conformance testing for a host or a router in network layer. In such model, the environment plays two roles: transport layer and link layer. That means, the environment can send or receive either the PDU from transport layer or the PDU from link layer. Consequently, this model allows users to provide the requests of transport layer and the services of link layer to stimulate the system to perform the conformance testing.

In order to perform interoperability testing between routers and hosts, we have to model networks. These networks have to contain an intermediate router, a destination router, a source host and a destination host. To simplify the system, we abstracted the minimum configuration of the system shown in figure 5.8. With this minimum configuration, the environment becomes transport layer only. The connection between hosts and routers plays the role of the link layer. That means the environment can send or receive PDU from transport layer only. Consequently, this model allows users to provide the requests of transport layer to stimulate the system to perform the conformance testing.

Let us consider a scenario in which Host1 sends a packet to a Host3. In this scenario, Host1 is a source, it may send a packet, an error message or fragments depending on the sending packet; Host3 is a destination, it may receive a packet or fragments, then may send an error message to the source or PDU to its upper layer; Router1 is an intermediate router; Roter2 is a destination router. Therefore, this minimum configuration covers all scenarios that achieve the requirements of IPv6, ICMPv6, MLD and RIPng protocols. This section has described the extracted technical information and minimum configuration. Now we are ready to create an SDL system.

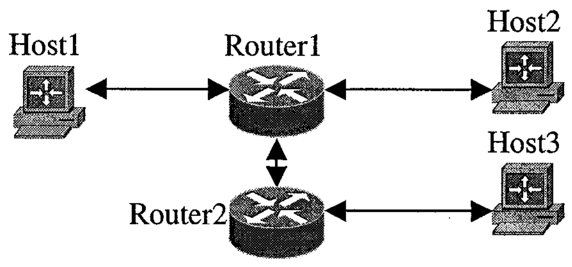


Figure 5.8 minimum configuration of the system

5.2 Creating an SDL Model

SDL sees the world as two parts: the system and its environment. The specification written in SDL is limited to the system. It is a formal model that defines the properties of an existing or planned system. Everything outside the system is a part of the environment. Even though the environment is not specified formally, it can be considered as a process, which can communicate with the system by signals.

This thesis considers network layer protocols IPv6, ICMPv6, MLD and RIPng. That means the SDL model considered here describes behaviours of these network layer protocols only. In order to perform conformance testing, the environment of the system plays the roles of transport layer and link layer. Therefore, the network layer is the top level of this model. Figure 5.9 is a graphic view of this concept.

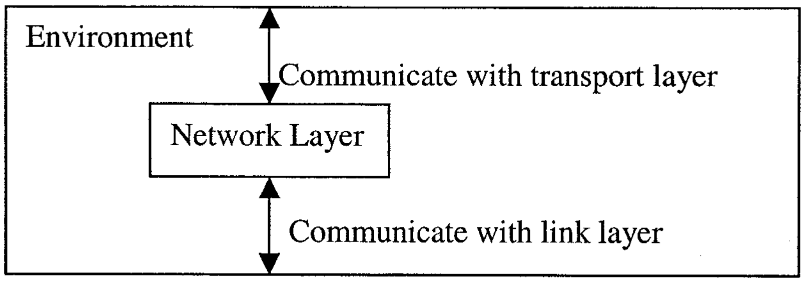
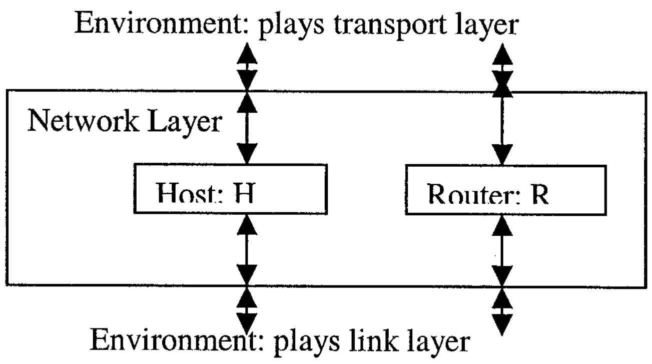


Figure 5.9 overview of the model for conformance testing

SDL provides hierarchical structure to decompose a system into blocks. Blocks are functional components used to simplify the complexities of a system. They can be self-containing. They communicate using signals via channels. Blocks indirectly affect a system by constraining signals between process instances.

Therefore, the conceptual view of SDL model for conformance testing can be described in figure 5.10. It contains two blocks: H and R. H is an instance of block type host. R is an instance of block type router.



Environment: plays transport layer
Network Layer
Host: H
Router: R
Environment: plays link layer

Figure 5.10 conceptual view of SDL model for conformance testing

In order to perform interoperability testing, another model is designed. This model's first layer's conceptual view is described in figure 5.11.

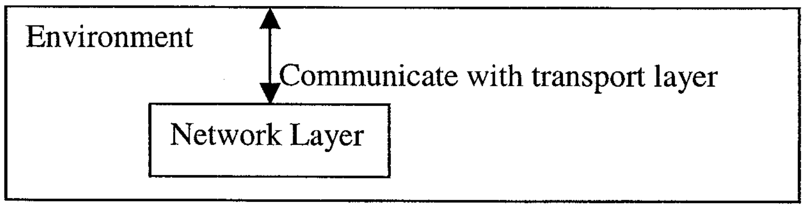


Figure 5.11 overview of the model for interoperability testing

The conceptual view of SDL model's second layer for interoperability testing can be described from the minimum configuration presented above in figure 5.8. It contains three hosts H1, H2 and H3, and two routers R1 and R2. It is described in figure 5.12.

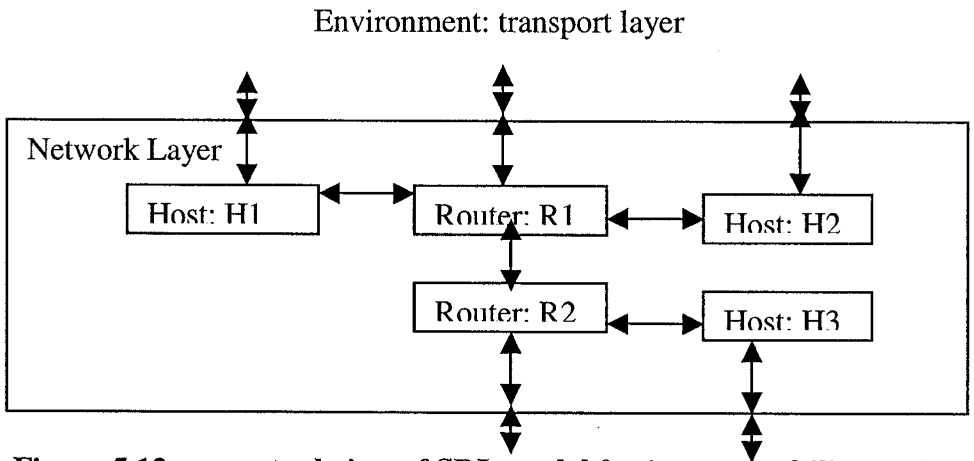


Figure 5.12 conceptual view of SDL model for interoperability testing

The architectures of systems are described. In order to simplify implementation and modification, this thesis uses object-oriented concept to classify two functional components: host and router. Therefore, the remaining task is to develop SDL model in ObjectGEODE.

5.2.1 SDL Decompositions

In order to simplify SDL models, there are two assumptions. We assume that a host has two connections: one connects to transport layer, the other one connects to a router. We also assume that a router has four connections: one connects to transport layer, one connects to a router, and two connect to hosts. With these two assumptions, SDL models are described.

Figure 5.13 is the SDL model for conformance testing.

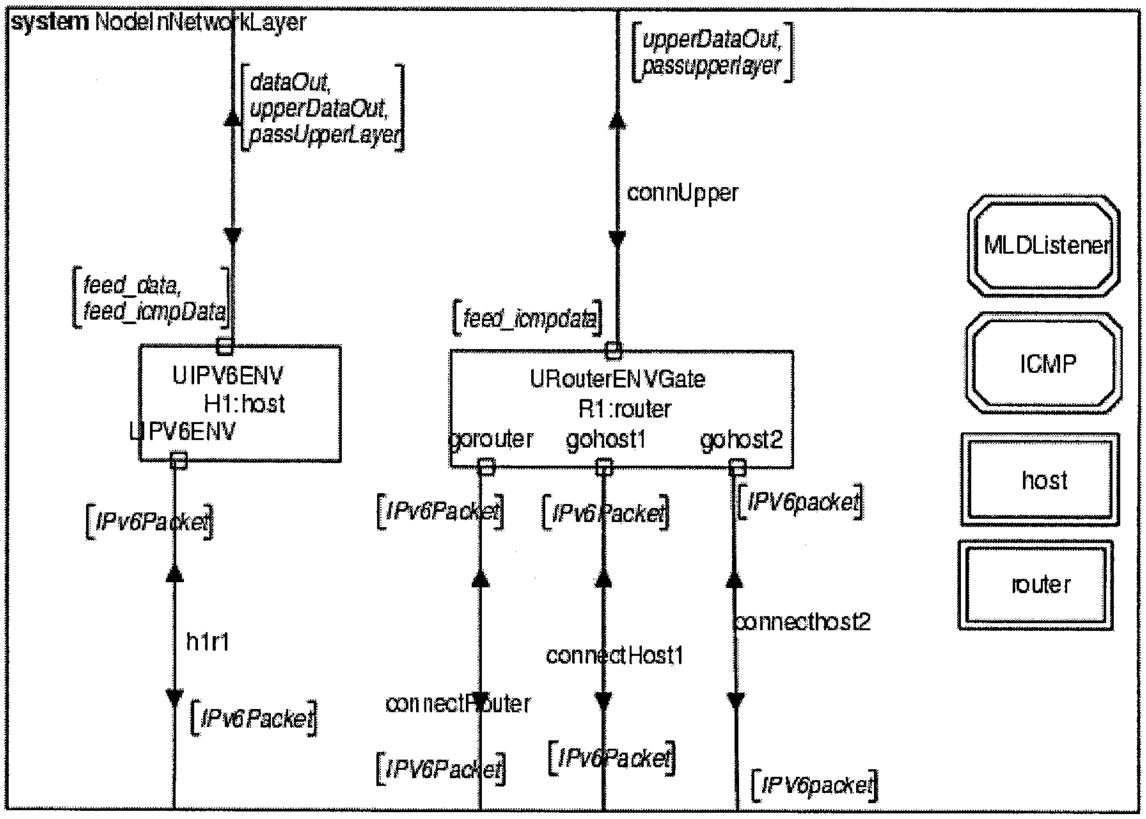


Figure 5.13 system interconnection diagram for conformance testing

This model is called NodeInNetworkLayer and contains two block instances: H1 and R1. H1 is an instance of block type host. R1 is an instance of block type router. In H1, gate UIPV6ENV connects to transport layer. It allows H1 to receive PDU from transport layer using signal feed_data or feed_icmpData, or send PDU to transport layer using signal dataOut, upperDataOut or passUpperLayer. Gate LIPV6ENV connects to link layer. It allows H1 to receive an IPv6 Packet from or send to link layer using signal IPV6Packet. In R1, gate URouterENVgate connects to transport layer. It allows R1 to receive PDU from transport layer using signal feed_icmpData, or send PDU to transport layer using signal upperlayerdata or passUpperlayer. Gate goRouter, ogHost1 and goHost2 connect to link layer, they allow the receiving of an IPv6 packet from or sending an IPv6 packet to link layer.

In consideration of the ICMP protocol implemented by every IPv6 node, process type ICMP, which describes the properties of ICMP protocol, is defined in system level. Consequently, both a host and a router can use process type ICMP.

MLD is an asymmetric protocol. It specifies different behaviours for multicast listener and for routers. For some multicast address, router itself can be a listener. In such case, the router performs both parts of the protocol. Therefore, process MLDListener, which describes the behaviours of multicast listener, is defined in system level.

Figure 5.14 is the SDL model for interoperability test.

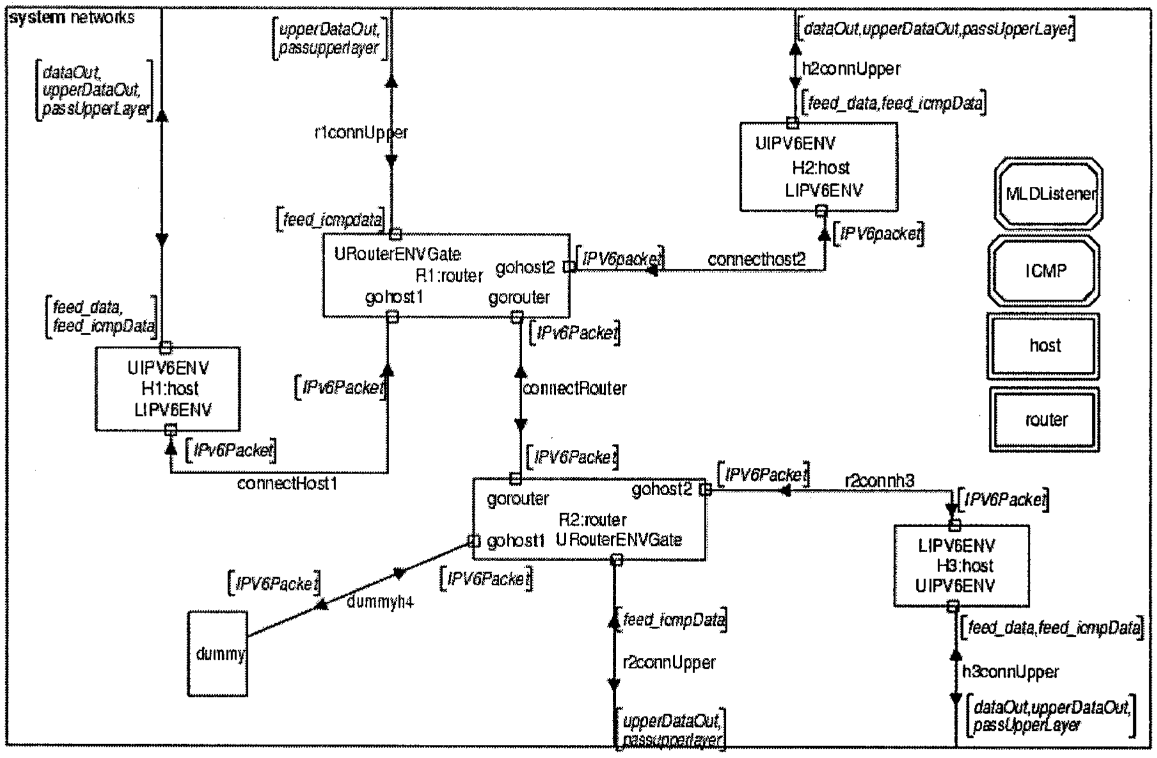


Figure 5.14 system interconnection diagram for interoperability testing

This system contains a dummy block defined to satisfy SDL semantics and five block instances: two routers (R1 and R2) and three hosts (H1, H2 and H3). R1 and R2 are instances of block type router. H1, H2 and H3 are block type host. When comparing this model with the model in figure 5.13, one can notice a conceptual change. Block type host connects to a router through gate LIPV6ENV to perform the communication with the router. In block type router, gate gohost1 and gohost2 permit connection to hosts to perform the communication with hosts. In block type router, Gate goRouter permit connection to a router to perform the communication with a router. Consequently, a user cannot communicate with this model as a link layer. This model simulates networks in real world with minimum configuration.

In this section, the system interconnection diagrams for both conformance testing and interoperability testing are described, the remaining task is to decompose block type host and router.

5.2.1.1 The Decomposition of Block Type Host

This block type contains three processes: IPV6Host, MLDHost and ICMPHost. Process IPV6Host implements the behaviours of IPv6 protocol for a host. Process MLDHost, which is an instance of process type MLDListener, describes the properties of MLD protocol for a host. Process ICMPHost, which is an instance of process type ICMP, describes the properties of ICMP protocol. Generally, the host block receives or sends an IPV6 packet through gate LIPV6ENV. It receives or sends PUD through gate UIPV6ENV. The specification of this block type is shown in figure 5.15.

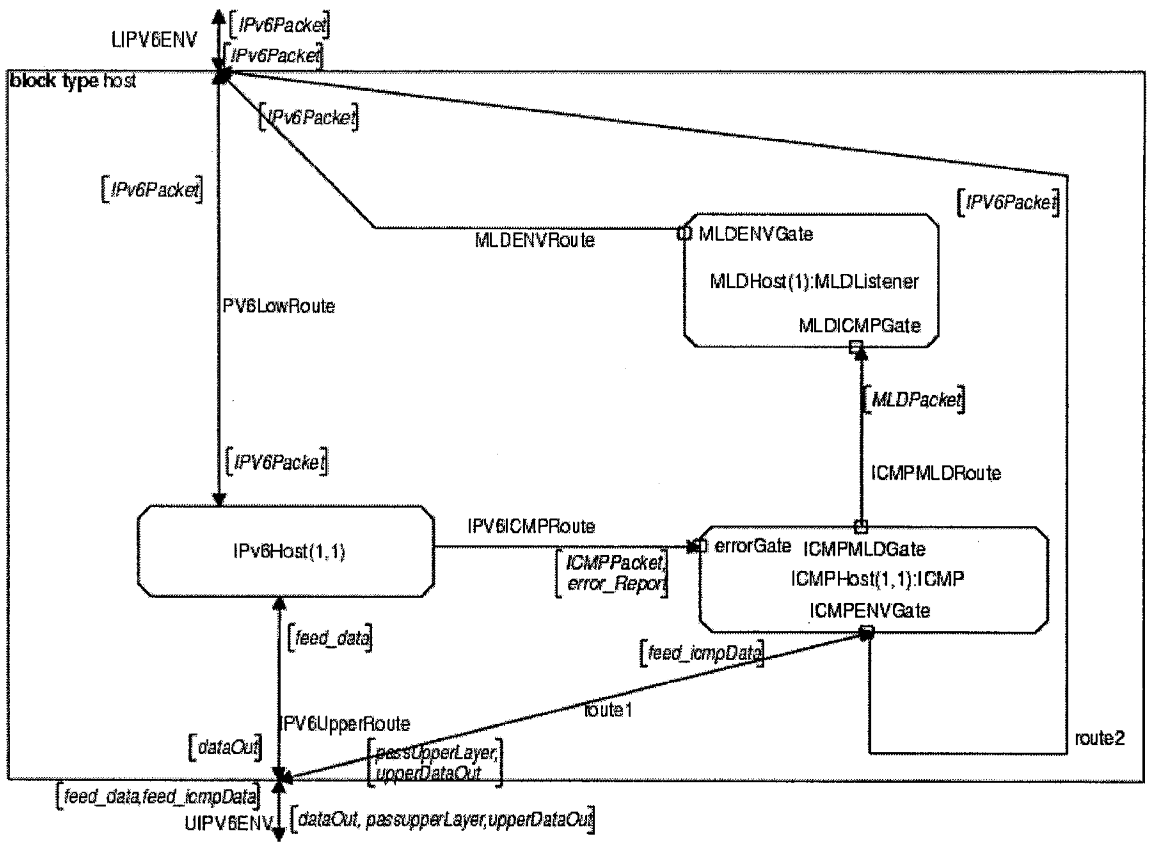


Figure 5.15 Interconnection diagram of block type host

5.2.1.2 The Decomposition of Block Type Router

This block type contains four processes: IPV6Router, ICMPRouter, RIPng and MLDQuerier. Process IPV6Router describes the properties of IPV6 protocol for a router. Process ICMPRouter is an instance of process type ICMP. Process RIPng describes the properties of RIPng protocol. Process MLDQuerier describes the properties of MLD protocol for a router. Generally, the router block receives or sends an IPv6 packet through three gates: goRouter, goHost1 and goHost2. it receives and sends PDU through gate URouterENVGate. The specification of this block type is shown in figure 5.16.

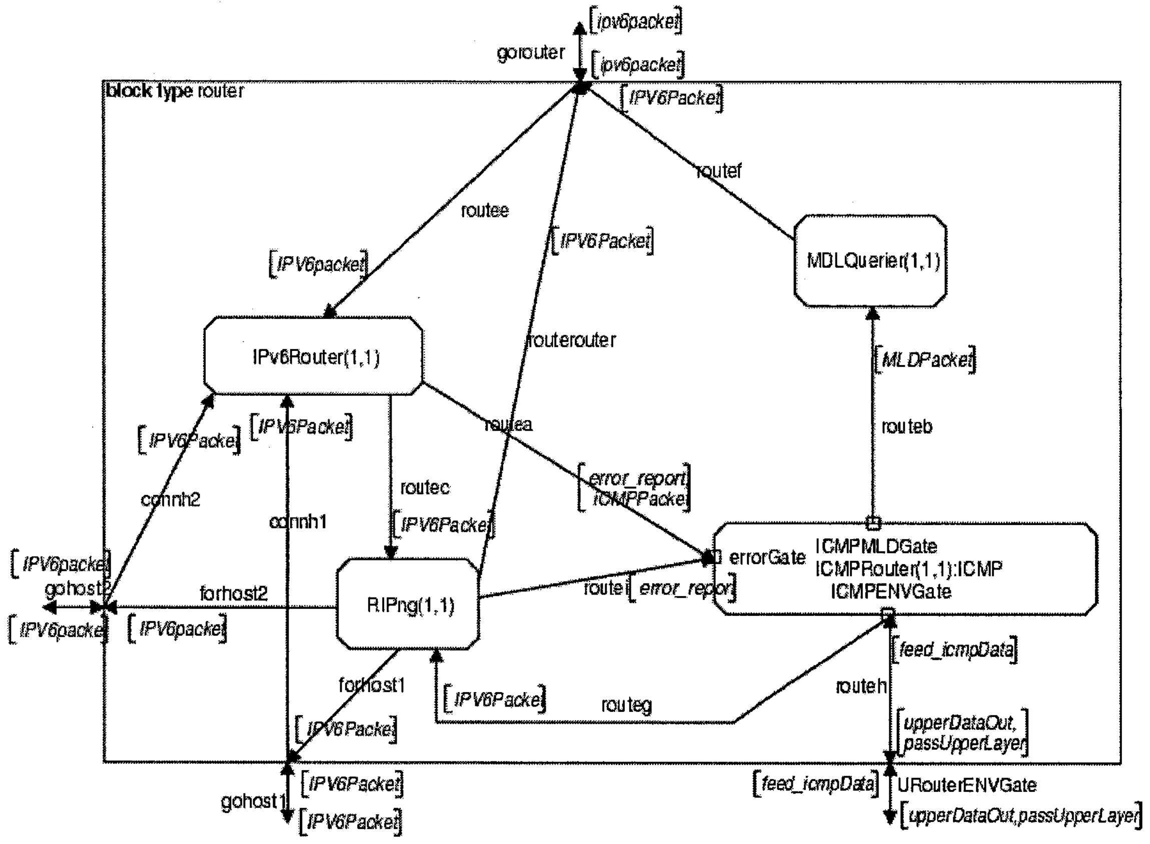


Figure 5.16 Interconnection diagram of block type router

5.2.2 Data Structure and Signals

This thesis designed the data structure by using abstract data type (ADT) and ASN.1 data type. The data structure hides data information and simplifies the implementation. Some data are scoped in the package in order to make the data consistent and reusable for the IP stack. The type IPv6Packet is one of the examples. The following is a detailed definition:

newtype IPv6Packet

struct

```
    IPv6Header IPv6Hdr;  /* IPv6 header */
    extenData extenHeader ;
    wrapData contents; /* all possible data. */
endnewtype IPv6Packet;
```

The 'IPv6Hdr' is a general structure in ADT, which includes only basic predefined data types. The type 'contents' and 'extenHeader' are two kinds of ASN.1 types. Their detailed definitions are as follows:

```
contents ::= CHOICE
 {
      ICMPErrorData   ICMPErrMsg,  /* carry ICMP error message. */
  ICMPInfoData   ICMPInfoMsg,  /* carry ICMP information message. */
  MLDData  MLDMsg,  /* carry MLD message. */
 RIPngPacket RIPngMsg, /* carry RIPng message */
 appData  charstring      /* other type data need to be wrapped in   IPPacket. */
 };
extenHeader ::= SEQUENCE
    {
         hopbyhop OptionHdr  optional , /* hop-by-hop option header. */
destOptHeader1  OptionHdr  optional  ,  /*  destination  option  header,  appears  in  the
routing table. */
    routingHeader RoutingHdr optional , /* routing header. */
    fragmentHeader FragmentHdr optional , /* fragment header. */
    destOptHeader2 OptionHdr optional /* destination option header.*/
};
```

In the general point of view, any data type can be declared as a parameter of a signal. A signal can have zero, one or more parameters. It carries these parameters to perform communication between two components.

Until now, we have described system decomposition, defined data structure, timers, signals and their parameters. It is time to write the SDL code.

5.2.3 Writing SDL Code

In SDL, the behaviour of a system is the joint behaviours of all process instances contained in the system. An SDL process instance may communicate both with each other and with environment through signals. It is a communicating extended finite-state machine (EFSM). It is either in a stable state or in a transition between states. A transition is triggered by the process instance receiving an input signal. Moreover, SDL allows the use of data and variables in a process. When the value of a variable in a process is changed, the global state of the process is changed. Therefore, the behaviour of the process may change. Consequently, writing SDL code is a task that describes EFSM in SDL graphic format supported by SDL Editor tool in ObjectGEODE. A procedure is a state machine within a process. It can be used to hide detailed information or repeat the same operation easily. As a result, defining a procedure may take place during a process' specification.

In order to specify a process efficiently, it is very helpful to draw an automata graph manually according to the extracted technical information, then refine and extend SDL code accordingly. The SDL code has to satisfy the SDL semantics.

The following is an example of the procedure of writing SDL code of process IPv6Host in the host block type. Figure 5.17 is the automata graph. At the beginning, a host is in its initial state. When it receives PDU data, which is shown in label L1, it changes to the state 'Form packet'. In this state, the host either sends a packet (packet' size <= MTU) or several packets (the original packet's size > MTU), which is shown in label L2, and then it changes to the initial state or the host changes to state 'Report error' by the output of an IPv6 packet. In this state, it generates an error message and sends this error message. Then it changes back to its initial state. When a host receives an IPv6 packet, which is shown in label L5, it changes to the state 'verify headers'. In this state, it verifies all headers included in the received packet. If any error has to be reported to the source node, it goes to the state 'report error', otherwise it goes to the state 'Get PDU'. In this state, the host removes IPv6 header and outputs PDU. Then it changes back to the initial state. We use this automata graph to develop EFMS for process IPV6Host in SDL graphic Editor. The SDL code has to satisfy SDL semantics at this step.

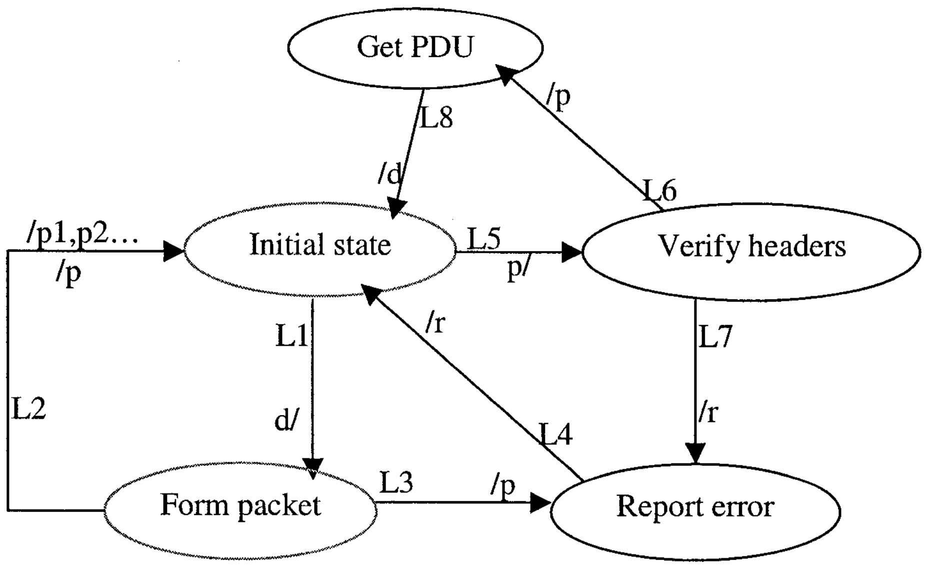


Figure 5.17 Automata for IPv6Host process

5.3 Verifying and Validating the SDL Model

Verification and validation are corrective methods that aim to detect errors and make corrections. Verification is a process of determining if the model is built correctly. First, the model has to satisfy the SDL static semantic rules, such as correct balance of the channel refinement, use of signals match to their declaration, use of data match to their declaration. To do so the designer uses Quick Checker in SDL Editor tool. Once the model is correct from the Quick Checker's point of view, the SDL simulator can execute the model. During the execution, the model is seen as a white box and the simulator acts as a debugger. In this step, all relevant errors, such as deadlocks, livelocks and unspecified receptions [7], should be captured and corrected. In order to avoid any new error being introduced, the model has to be verified repeatedly. Verification is complicated and time-consuming. However, it is simplified by using SDL simulator. During the execution, it is easy to locate the errors by tracing on the process states, queues and variables. Moreover, the cover rates on process transition can point out how many transitions have been covered, which helps to refine input data.

Validation aims at knowing if the model is correct [7]. It means that validation assures that the model was built to satisfy the related RFCs. In order to validate the model, a set of MSC scenarios has to be specified for the main behaviors of extracted technical information. To validate the model under a certain behavior, one has to execute the related MSC scenario file and the SDL code together in SDL simulator. During this execution, each MSC observes the behaviors of the model and compares them with the expected behaviors. Any observed error would be reported by the simulator. SDL code can only compile with one MSC file at a time.

Once the SDL model passes the syntax checker (Quick Check in SDL Editor), the SDL model can be simulated with the ObjectGEODE interactive simulator. The SDL model is opened and can be seen as a white box. The user plays the role of environment by sending the model's input and by controlling timer. In order to send input to the model, a set of input data have to be derived by combining sequence diagrams and SDL implementation codes. These input data will be refined during the simulation.

The principle of designing input data is that each input data should focus on a single or related set of conformance requirements. As a white box test, the designed data should try to execute as many different branches as possible, in a process or a procedure. Consequently, the more branches are executed, the more the possible behaviours appear, and the higher the transition cover rate will be.

The environment sends data to an SDL model by using 'feed' command, which is performed by compiling the SDL model with a MSC file. In verification stage, the MSC file defines input data only. In validation stage, the MSC define both input data and output data. Once the environment stimulates the SDL model with the input data, verification and validation takes place in the interactive simulation model. Figure 5.18 is an on-fly MSC generated during simulation. Using this MSC file, we can trace the communication between components in the system under implementation. By doing this, it is much easier to verify the implementation and consult logical errors.

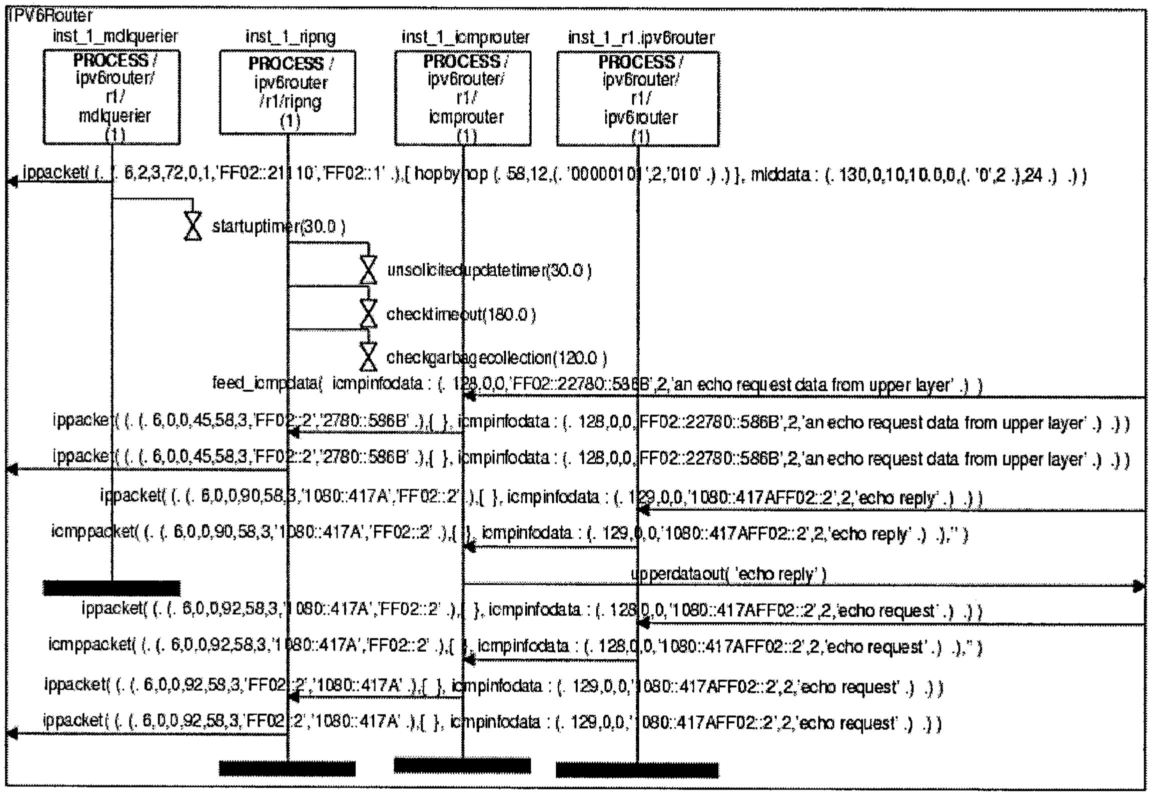

IPV6Router
inst_1_mdlquerier
inst_1_ripng
inst_1_icmprouter
inst_1_r1.ipv6router
PROCESS / ipv6router/ r1/ mdlquerier (1)
PROCESS / ipv6router /r1/ripng (1)
PROCESS / ipv6router/ r1/ icmprouter (1)
PROCESS / ipv6router/ r1/ ipv6router (1)
ippacket((. (. 6,2,3,72,0,1,'FF02::21 10','FF02::1' .),[hopbyhop (. 58,12,(. '00000101',2,'010' .) .)], mlddata : (. 130,0,10,10.0,0,(. '0',2 .),24 .) .))
startuptimer(30.0)
unsolicitedupdatetimer(30.0)
checktimeout(180.0)
checkgarbagecollection(120.0)
feed_icmpdata(icmpinfodata : (. 128,0,0,'FF02::22780::586B',2,'an echo request data from upper layer' .))
ippacket((. (. 6,0,0,45,58,3,'FF02::2','2780::586B' .),{ }, icmpinfodata : (. 128,0,0,FF02::22780::586B',2,'an echo request data from upper layer' .) .))
ippacket((. (. 6,0,0,45,58,3,'FF02::2','2780::586B' .),{ }, icmpinfodata : (. 128,0,0,FF02::22780::586B',2,'an echo request data from upper layer' .) .))
ippacket((. (. 6,0,0,90,58,3,'1080::417A','FF02::2' .),{ }, icmpinfodata : (. 129,0,0,'1080::417AFF02::2',2,'echo reply' .) .))
icmppacket((. (. 6,0,0,90,58,3,'1080::417A','FF02::2' .),{ }, icmpinfodata : (. 129,0,0,'1080::417AFF02::2',2,'echo reply' .) .),'')
upperdataout('echo reply')
ippacket((. (. 6,0,0,92,58,3,'1080::417A','FF02::2' .),{ }, icmpinfodata : (. 128,0,0,'1080::417AFF02::2',2,'echo request' .) .))
icmppacket((. (. 6,0,0,92,58,3,'1080::417A','FF02::2' .),{ }, icmpinfodata : (. 128,0,0,'1080::417AFF02::2',2,'echo request' .) .),'')
ippacket((. (. 6,0,0,92,58,3,'FF02::2','1080::417A' .),{ }, icmpinfodata : (. 129,0,0,'1080::417AFF02::2',2,'echo request' .) .))
ippacket((. (. 6,0,0,92,58,3,'FF02::2','1080::417A' .),{ }, icmpinfodata : (. 129,0,0,'1080::417AFF02::2',2,'echo request' .) .))

Figure 5.18 On-fly MSC

Once the system has been proven correct, it is ready to generate test cases. Section 5.4 describes the principle of test generation.

5.4 Test Generation

The aim of test generation is to produce test suites. TestComposer generator is used to perform test generation. Generally, it works in three phases [8]:

1). Test purpose design phase: test purpose can be either computed automatically or written by the user,

2). Test case generation phase: each test purpose is entered in the test case generation engine, the generated test case being stored into the test case database,

3). Test suite production phase: a production model uses the database through an API, and generates the test suite in a user-define format.

These three phases are depicted in figure. 5.19.

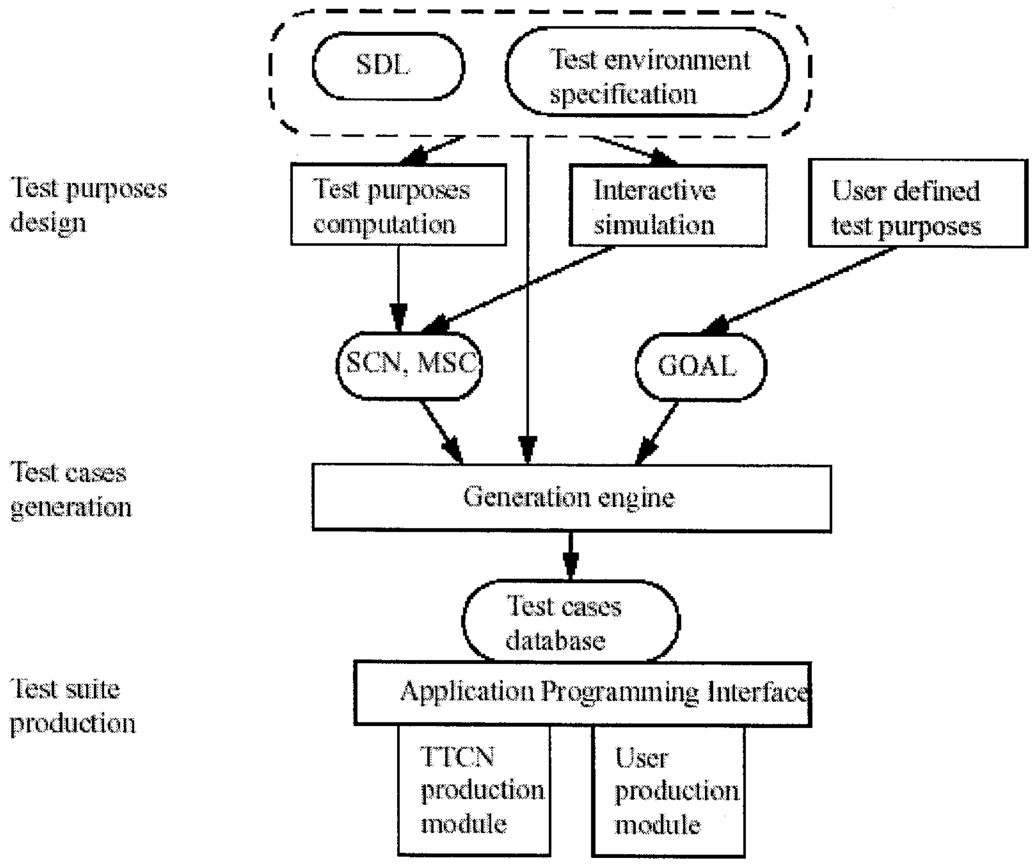


Figure 5.19 general functional architecture

As described in figure 5.19, TestComposer takes SDL specification and test environment specification to produce test suites.

SDL specification describes the system under test (SUT), which includes the description of the implementation under test (IUT), and of the test architecture.

IUT is a given implementation on which the test suites will be applied. It is the first part of the SDL specification to be built.

The test architecture is constituted by points of controls and observation (PCO) together with service access points (ASP). PCO corresponds directly to a channel. It is a gate through which the tester can communicate with the system under test. ASP is a special interface used to access IUT, because it happens very often that IUT cannot be accessed directly.

The test environment can be modeled using either 'output' instruction, which sends a signal once, from environment to system; or 'feed' instruction, which describes the kind of signals that the environment is ready to send to the system at any time.

There are two advantages to the use of 'feed' command. The first one is that it can avoid a complicated manual typing when using 'output' command to send data from environment to the system under test. The other is that it can avoid corresponding test case errors in TTCN editor, such as 'incorrect channel define', 'the channel is impossible to receive this kind of UDP', or 'the channel is impossible to send the packet'. All these kinds of errors will occur during the analysis of the test case in TTCN editor if 'output' command has been used. The disadvantage of using 'feed' command is that any message can be sent at any moment. If any unexpected message is sent, execution errors will occur. In order to solve this problem, the tester has to stimulate transactions properly during the simulation. In other words, the tester has to control message sending properly and accurately.

5.4.1 Test Purposes Generation

A test purpose is some abstract properties that the system must have. It corresponds to the answer to the "What do I want to test?" question. It is used for selecting different behaviours described in the SDL specification. The selected behaviours will be used as a basis for defining a test case. In ObjectGEODE, a test purpose can be created manually by using SDL & MSC Editor, or automatically during the test purpose generation phase by using testComposer. During the test purpose generation, the structural coverage of the SDL specification is computed for analysis.

Test coverage, which is called structural coverage, identifies the parts of the SUT actually tested by the generated test cases. The coverage is computed from the beginning of the test, at test purpose level. Basically, each test purpose corresponds to a set of a behavioural paths identified in the specification. The structural entities entered along these behavioural paths can be said to be covered, and thus meet the test purpose.

Each computed test purpose involves one observation step, which is a sequence of basic blocks that lead from one stable testing state to another stable testing state. A stable testing state is defined as an SUT state in which the only possible actions are the input of an observable test event, or the timeout of an internal timer.

A basic block is a coverage unit. It begins with start, input, priority input, continuous signal or an answer to a decision. It ends with a nextstate, a join, a stop or the return instruction of a procedure.

The coverage defined in a test purpose is expressed as the set of basic blocks of the SDL specification it allows to be covered. Each test purpose corresponds to the coverage of one or several blocks.

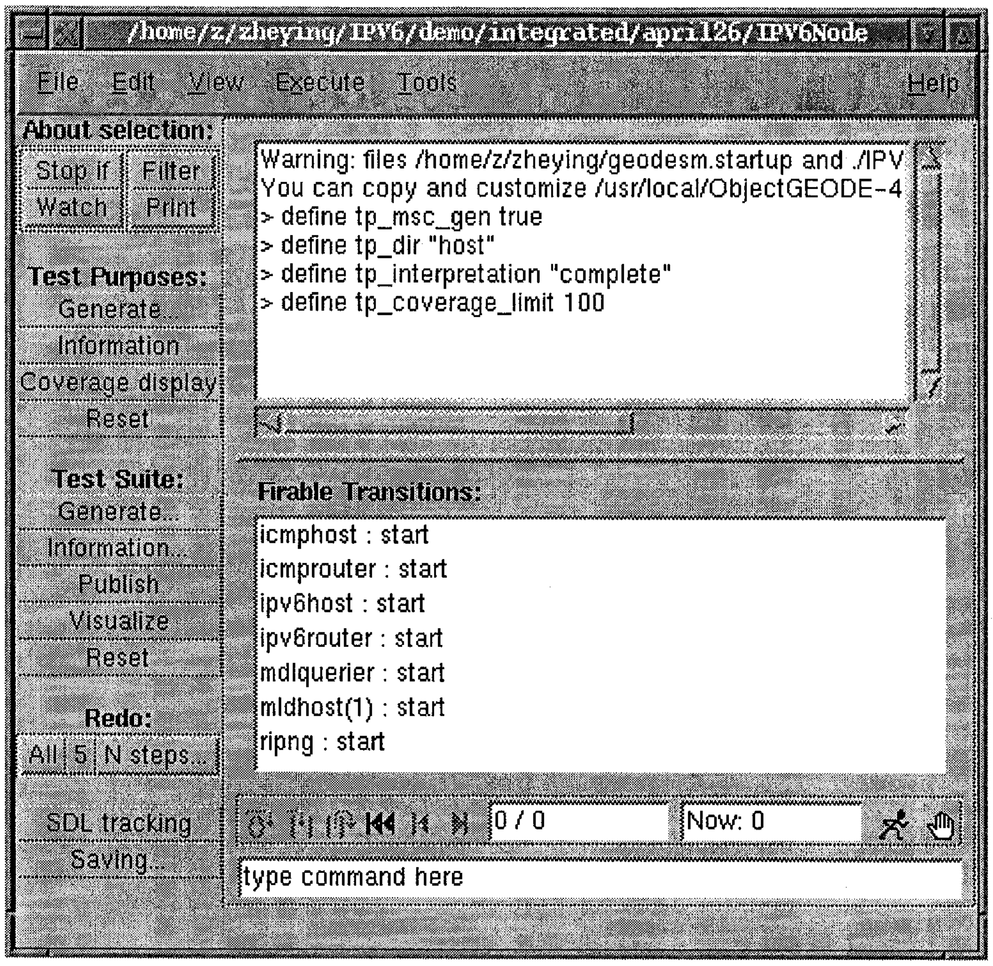
Figure 5.20 is the main window of ‘TestComposer’, an interface through which a tester can generate test purpose and test cases automatically. A tester can easily describe its environment in the command window. The detailed description is as follows:

1). “define tp_msc_gen true”, which requests to generate a test purpose when a scenario file is generated;

2). “define tp_dir ‘testpurposeFolder’ ”, which sets generated test purposes to be saved in folder ‘testpurposeFolder’;

3) “define tp_interpretation ‘complete’ ”, which specifies the completion of the interpretation mode associated to a generated test purpose;

4) “define tp_coverage_limit 100”, which specifies the aim coverage to be reached during test purpose generation.



```
Warning: files /home/z/zheying/geodesm.startup and ./IPV
You can copy and customize /usr/local/ObjectGEODE-4
> define tp_msc_gen true
> define tp_dir "host"
> define tp_interpretation "complete"
> define tp_coverage_limit 100
```

```
icmphost : start
icmprouter : start
ipv6host : start
ipv6router : start
mdlquerier : start
mldhost(1) : start
ripng : start
```

```
type command here
```

Figure 5.20 main window of 'TestComposer'

As shown in figure 5.20, the 'firable' transitions can be fired in 'firable transitions' window. After executing all necessary 'fireable' transitions, the test purpose can be generated by clicking 'Generate Current Button' in test purpose command interface.

5.4.2 Test Cases Generation

A test case corresponds to a test that is run on the SUT. It is an operational view of the test. It contains all necessary information such as all expected events, timers setting and test verdicts. In other words, it describes what are the expected events leading to the test success, but also reactions to not expected events, as well as control of the time one should wait for an expected event. It corresponds to the answer to the "How do I test?" question.

A test case can be computed in TestComposer by taking a test purpose and a SDL specification and is stored in a test case database in an internal format.

The most important elements are 'behavior description', 'constraints Ref' and 'verdict'. The behavior description shows information about 'input signal', 'output signal', related PCOs and timers. Some timers, which are called test timers, are automatically generated in the test cases. There are four kinds of test timers:

1). TAC: it is set when the tester is waiting for a SUT output event. If there is a TAC times out, test verdict should be FAIL.

2). TNOAC: it is set to detect an output block. No verdict is associated with a TNOAC times out.

3). TWAIT: it measures the maximum time to execute an implicit sending. A FAIL verdict is related to a TWAIT times out.

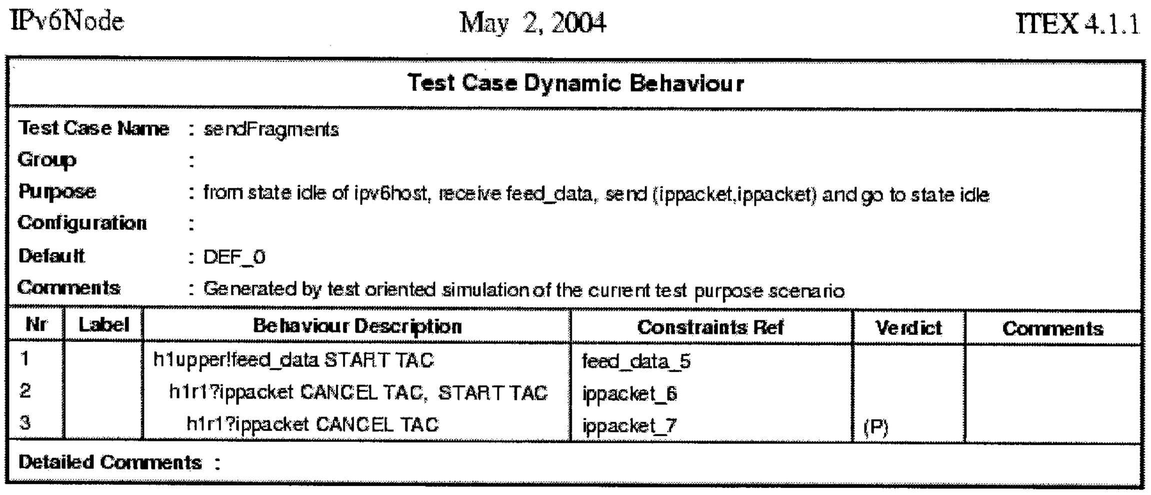
4). TEMPTY: it measures the maximum time for expiration of specification timer. No verdict is related to a TEMPTY times out.

The 'constraints Ref' are references of input signals and output signals. According to these signals, one can get the values of parameters of these signals in 'constraints definition part' of TTCN editor. The verdict is related to the different conclusions of a test case. The PASS verdict corresponds to the successful execution of a test case. The INC (inconclusive) verdict corresponds to a valid outcome, described in the test case, but not relevant to the test purpose corresponding to this test case. The FAIL verdict corresponds to an invalid behavior such as an unexpected message, or the expiration of some of the test timers.

Figure 5.21 is a test case in TTCN GR format. The test case presents a source sending two fragments after it received data.

The principle of the execution is that a source can only send a packet whose size is less than or equal to MTU. Consequently, if the packet is too big to send, the source must fragment the packet.

The first entity shows that the environment inputs the data, which is carried by signal feed_data through PCO h1upper. The actual value of the input is referred to feed_data_5 in constrains part. Meanwhile TAC timer is set for waiting the output. The second entity shows the system sends output, which is carried by signal ippacket through PCO h1r1. The actual value is referred to ippacket_6 in constrains part. At this moment, the timer TAC is cancel (means the output is sent before the TAC times out), and other TAC is set to wait for another output. The third entity shows the system sends output, which is carried by signal ippacket through PCO h1r1. The actual value is referred to ippacket_7 in constrains part. At this moment, the timer TAC is cancel (means the output is sent before the TAC times out), and no more TAC is set; that means that there is no more output. As a result, the test verdict P indicates the test case is successful.



IPv6Node May 2, 2004 ITEX 4.1.1

Test Case Dynamic Behaviour

Test Case Name	: sendFragments
Group	:
Purpose	: from state idle of ipv6host, receive feed_data, send (ippacket,ippacket) and go to state idle
Configuration	:
Default	: DEF_0
Comments	: Generated by test oriented simulation of the current test purpose scenario

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		h1upper!feed_data START TAC	feed_data_5		
2		h1r1?ippacket CANCEL TAC, START TAC	ippacket_6		
3		h1r1?ippacket CANCEL TAC	ippacket_7	(P)	

Detailed Comments :

Figure 5.21 test case in TTCN GR format

5.4.3 Test Suites Description

A test suite is a collection of test cases with all necessary declarations such as: data type, timer, and signals. It can be structured by grouping test cases into categories.

The generated test cases are stored into a database in TestComposer generation tool. This database is accessible through an Application Programming Interface. In the test suite graphical interface, 'publish button' translates the internal test cases database into a file. The format of this file depends on the selected production library. In our case, the file is in TTCN-MP format. A MP file is not readable, a dedicated TTCN editor should be used to edit and visualize this format.

A TTCN suite includes test suite overview, declaration part, constraints part, and dynamic part. The test overview consists of test suite structure, which gives general information on the test suites, test case index, which lists all the test cases of the suite, test step index, which lists all the test steps of the suite, and default index, which lists the only default test case generated by TestComposer. In our case, test suite structure and test step index are empty. Default index equals to DEF_0.

The declaration part includes all type definitions generated by TestComposer. The type definitions are global to the test suite. All SDL data types and ASN.1 declarations are translated into ASN.1 definitions compliant with the TTCN standards. The detail description of ASN.1 type definition, PCO type declarations, timer declarations and ASN.1 PDU type definitions can be viewed in this part.

The constraints part includes PDU constrains that describes ASN.1 PDU constraint declarations, which are related packets for modeling test environment.

Dynamic part includes test cases, test step library and default library. The test cases are the most interesting part. In this part, individual test case can be reviewed.

6. Conclusions

The current project presents an overview of modeling, specifying and testing the main protocols in IPv6 stack, such as IPv6, ICMPv6, RIPng, and MLD. It describes the general methodology considered to obtain formal specification, test purpose and TTCN test cases. It also described the rational behind the modeling. The project produces an SDL specification, accomplishes conformance testing and interoperability testing, and generates a TTCN test suite for IPv6 protocols. From the generated TTCN test suite one can manually generate executable test cases in C and execute them in an active environment called FLAME [12]. In this active environment, test cases are loaded into a "Tester Node" that will execute the test case and report the observed result.

The conformance and interoperability tests are produced according to information extracted from the RFCs. RFCs are difficult to understand due to their ambiguity, their incompleteness and multiple self-references (one RFC often refers to other RFCs that refers to each other and to the first RFC). Moreover, RFCs are not written in formal language. As a result, it is very difficulty to model RFCs in a formal language, which gives very precise and complete description.

According to the experience of modeling IPv6 protocols, SDL is suitable to describe IPv6 protocols. With graphic presentation of SDL, the static and dynamic behaviour of the system can be described at the same time. Moreover, it is easy to define an IPv6 packet with SDL abstract data type concept.

Data type definition is inconsistent between SDL and TTCN notation. SDL is formal, object-oriented, and standardized. It allows a data type definition to exist in different objects with the same name. However, data type definitions are global in the test suite of TTCN. Therefore, TTCN only allows data type definitions that are global to the test suite. In order to be consistent with TTCN, global data types have to be defined in the SDL model. Therefore, SDL loses its object-oriented specificity. In addition, in ObjectGEODE, the description of test environment is inconsistent because there are two instructions to describe the test environment: ‘output’ instruction and ‘feed’ instruction. These two instructions give different test cases and only the test environment described using ‘feed’ instruction is consistent with TTCN. This might be caused by the TestComposer type definition translation mode or by the standards themselves.

In order to accelerate and expand IPv6’s deployment, it is necessary to formalize all IPv6 protocols and generate corresponding test cases.

References

[1] WEB Documents, www.tahi.org

[2] WEB Documents, www.ispras.ru/~RedVerst

[3] S. Deering, R. Hinden, RFC 2460, Internet Protocol Version 6 (IPv6) Specification, December 1998

[4] A. Conta, S. Deering, RFC 2463, Internet Control Message Protocol (ICMPv6) for Internet Protocol Version 6 (IPv6) Specification, December 1998

[5] G. Malkin, R. Minnear, RFC 2080, RIPng for IPv6, January 1997

[6] ObjectGEODE 4.2 Document, tutorial, October 2000

[7] ObjectGEODE 4.2 Document, SDL Simulator-Reference Manual, October 2000

[8] ObjectGEODE 4.2 Document, TestComposer –Reference Manual, October 2000

[9] WEB Documents, www.iec.org/online/tutorials/ttcn/

[10] Gregor v. Bochmann, Protocol Specification for OSI, Computer Networks and ISDN Systems 18 (1989/90) 167-184

[11] ITU-T Recommendation Z.100 (11/99)

[12] Abdelghani Benharref, Rachida Dssouli, Zineb Berbich, Formal Specification, TTCN and Executable Test cases for Main IPv6 Protocols, The 2nd IEEE International Symposium On Signal Processing and Information Technology, P348

[13] F. Belina & D. Hogrefe, the CCITT Specification and Description Language SDL, Computer Networks and ISDN Systems, September 1989

[14] S. Deering, W. Fenner, B. Haberman, RFC 2710, Multicast Listener Discovery (MLD) for IPv6, October 1999

[15] J. Postel, ISI, RFC 768, User Datagram Protocol (UDP), 28, August 1980

[16] Web Documents: unitesk.ispras.ru/products/ctesk/tools/

Appendix A: Conformance Test Cases for Hosts

This section presents selected test purposes and their corresponding test cases for conformance testing for IPv6 hosts. An IPv6 host considered here should have properties of IPv6, ICMPv6 and MLD protocols. The selected test purposes should have proper input data, whose observable behaviours should be shown after it stimulates the SUT. The following are these selected test purposes and corresponding test cases.

A1: Sending and Receiving IPv6 Packets

This test purpose shows the basic behaviors of a source and a destination. In figure A1.1, both signal feed_data and dataout carry PDU of transport layer. Signal ippacket carries IPv6 packet. The first transition shows that the system receives a PDU from transport layer. The second transition shows that the system sends an IPv6 packet to link layer. The third transition shows that the system receives an IPv6 packet from link layer. The last transition is the system sending out PDU to transport layer. Figure A1.2 is the corresponding test case.

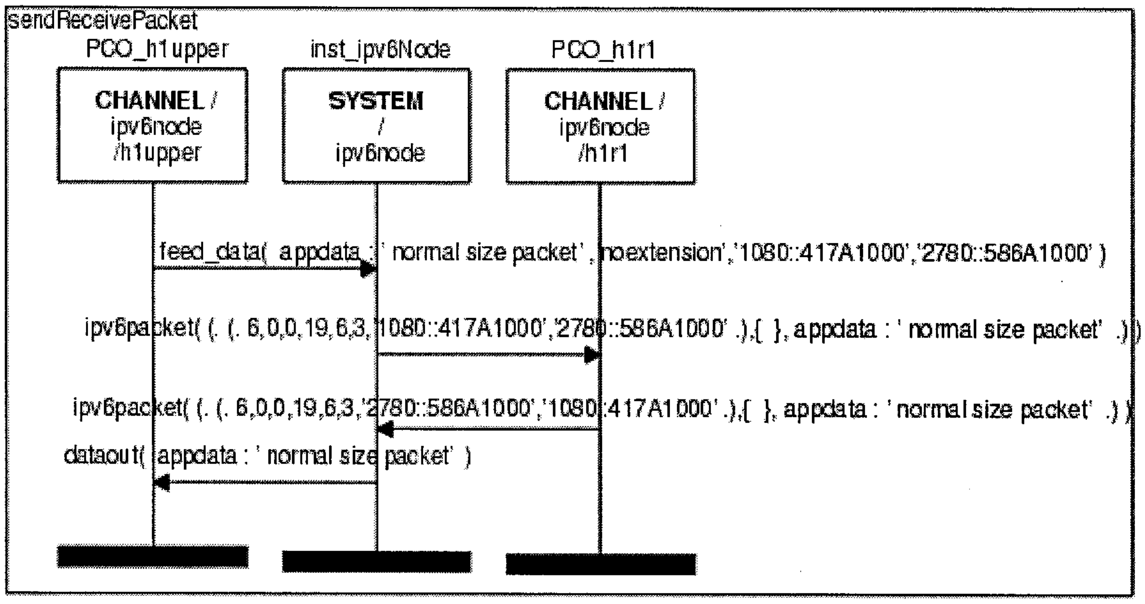

sendReceivePacket
PCO_h1upper
inst_ipv6Node
PCO_h1r1
CHANNEL / ipv6node /h1upper
SYSTEM / ipv6node
CHANNEL / ipv6node /h1r1
feed_data(appdata : ' normal size packet' , noextension','1080::417A1000','2780::586A1000')
ipv6packet((. (. 6,0,0,19,6,3,'1080::417A1000','2780::586A1000' .),{ }, appdata : ' normal size packet' .))
ipv6packet((. (. 6,0,0,19,6,3,'2780::586A1000','1080::417A1000' .),{ }, appdata : ' normal size packet' .))
dataout(appdata : ' normal size packet')

Figure A1.1 test purpose for sending and receiving IPv6 packets

IPv6Node May 2, 2004 ITEX 4.1.1

Test Case Dynamic Behaviour

Test Case Name	: sendReceivePacket
Group	:
Purpose	: from state idle of ipv6host, receive ippacket, send (dataout) and go to state idle
Configuration	:
Default	: DEF_0
Comments	: Generated by test oriented simulation of the current test purpose scenario

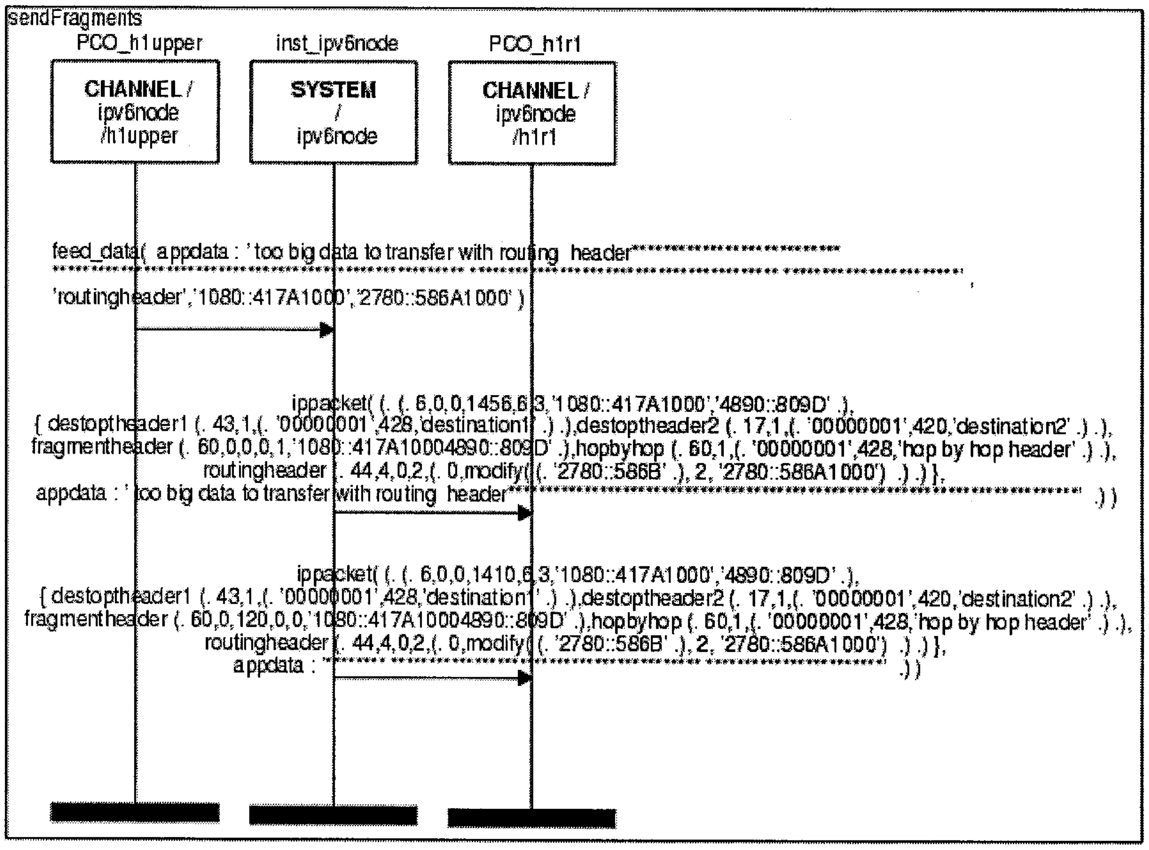
Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		h1upper!feed_data START TAC	feed_data_1		
2		h1r1?ippacket CANCEL TAC	ippacket_2		
3		h1r1!ippacket START TAC	ippacket_3		
4		h1upper?dataout CANCEL TAC	dataout_4	(P)	

Detailed Comments :

Figure A1.2 test case for sending and receiving IPv6 packets

A2: Sending Fragments

This test purpose shows a source sending fragments. In figure A2.1, the first transition shows the system receiving PDU from transition layer, and then system sending out two IPv6 packets to link layer, which are shown in the second and the third transition. Figure A2.2 is the corresponding test case.

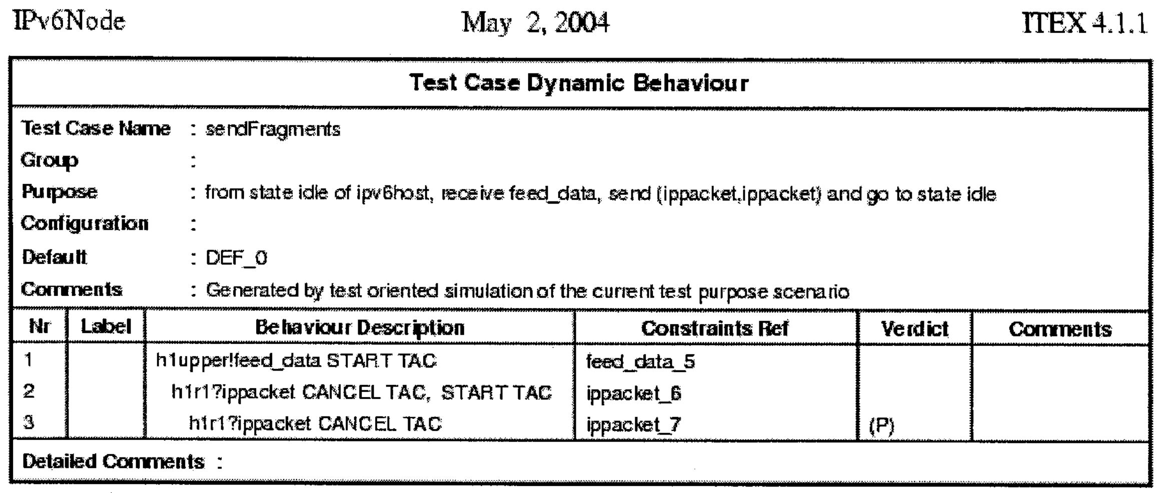


```
feed_data( appdata : 'too big data to transfer with routing header*****************************
********************************************************************************************
'routingheader','1080::417A1000','2780::586A1000' )

ippacket( (. (. 6,0,0,1456,63,'1080::417A1000','4890::809D' .),
{ destoptheader1 (. 43,1,(. '00000001',428,'destination1' .) .),destoptheader2 (. 17,1,(. '00000001',420,'destination2' .) .),
fragmentheader (. 60,0,0,0,1,'1080::417A10004890::809D' .),hopbyhop (. 60,1,(. '00000001',428,'hop by hop header' .) .),
routingheader (. 44,4,0,2,(. 0,modify((. '2780::586B' .), 2, '2780::586A1000') .) .)},
appdata : 'too big data to transfer with routing header**************************************************** .) )

ippacket( (. (. 6,0,0,1410,63,'1080::417A1000','4890::809D' .),
{ destoptheader1 (. 43,1,(. '00000001',428,'destination1' .) .),destoptheader2 (. 17,1,(. '00000001',420,'destination2' .) .),
fragmentheader (. 60,0,120,0,0,'1080::417A10004890::809D' .),hopbyhop (. 60,1,(. '00000001',428,'hop by hop header' .) .),
routingheader (. 44,4,0,2,(. 0,modify((. '2780::586B' .), 2, '2780::586A1000') .) .)},
appdata : '***************************************************************** .) )
```

Figure A2.1 test purpose for sending fragments



IPv6Node May 2, 2004 ITEX 4.1.1

Test Case Dynamic Behaviour

Test Case Name	: sendFragments
Group	:
Purpose	: from state idle of ipv6host, receive feed_data, send (ippacket,ippacket) and go to state idle
Configuration	:
Default	: DEF_0
Comments	: Generated by test oriented simulation of the current test purpose scenario

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		h1upper!feed_data START TAC	feed_data_5		
2		h1r1?ippacket CANCEL TAC, START TAC	ippacket_6		
3		h1r1?ippacket CANCEL TAC	ippacket_7	(P)	

Detailed Comments :

Figure A2.2 test case for sending fragments

A3: Receiving Fragments

This test purpose presents a destination that will not reassemble the original packet from its fragments until all fragments are received. No fragments should contain an error that causes the system to discards it. In figure A3.1, the first transition is the last fragment of a fragmented IPv6 packet, which is identified by M flag (the value is 0) in the fragment header. The second transition is the first fragment of a fragmented IPv6 packet, which is identified by offset field (the value is 0) in the fragment header. Therefore, the original IPv6 packet is reassembled. Consequently, the PDU carried by this original IPv6 packet is sent to the transport layer. This behavior is shown in the last transition. figure A3.2 is the corresponding test case.

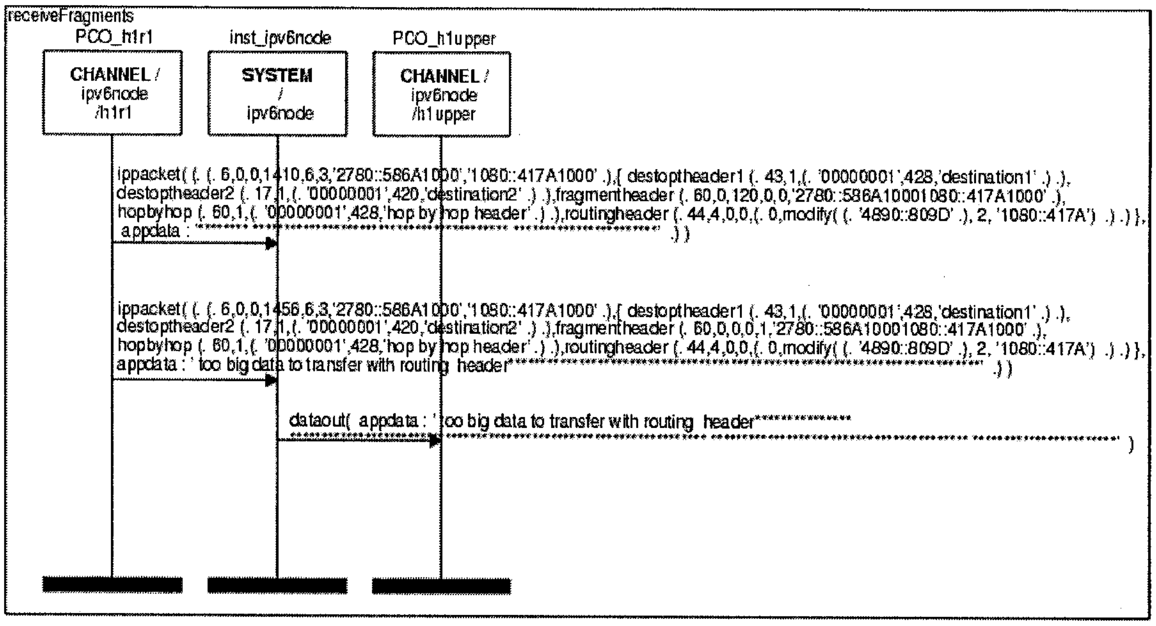

receiveFragments
PCO_h1r1
inst_ipv6node
PCO_h1upper
CHANNEL / ipv6node /h1r1
SYSTEM / ipv6node
CHANNEL / ipv6node /h1upper
ippacket((. (. 6,0,0,1410,6,3,'2780::586A1000','1080::417A1000' .),{ destoptheader1 (. 43,1,(. '00000001',428,'destination1' .) .), destoptheader2 (. 17,1,(. '00000001',420,'destination2' .) .),fragmentheader (. 60,0,120,0,0,'2780::586A10001080::417A1000' .), hopbyhop (. 60,1,(. '00000001',428,'hop by hop header' .) .),routingheader (. 44,4,0,0,(. 0,modify((. '4890::809D' .), 2, '1080::417A') .) .)}, appdata : '******************************' .))
ippacket((. (. 6,0,0,1456,6,3,'2780::586A1000','1080::417A1000' .),{ destoptheader1 (. 43,1,(. '00000001',428,'destination1' .) .), destoptheader2 (. 17,1,(. '00000001',420,'destination2' .) .),fragmentheader (. 60,0,0,0,1,'2780::586A10001080::417A1000' .), hopbyhop (. 60,1,(. '00000001',428,'hop by hop header' .) .),routingheader (. 44,4,0,0,(. 0,modify((. '4890::809D' .), 2, '1080::417A') .) .)}, appdata : ' too big data to transfer with routing header******************************' .))
dataout(appdata : ' too big data to transfer with routing header******************************')

Figure A3.1 test purpose for receiving fragments

IPv6Node	May 2, 2004	ITEX 4.1.1

Test Case Dynamic Behaviour

Test Case Name	: receiveFragments
Group	:
Purpose	: from state idle of ipv6host, receive ippacket, send (dataout) and go to state idle
Configuration	:
Default	: DEF_0
Comments	: Generated by test oriented simulation of the test purpose /home/z/zheying/IPV6/demo/integrated/receiveFragments.msc

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		h1r1!ippacket START TAC	ippacket_8		
2		h1r1!ippacket CANCEL TAC, START TAC	ippacket_9		End of preamble
3		h1upper?dataout CANCEL TAC	dataout_10	(P)	

Detailed Comments :

Figure A3.2 test case for receiving fragments

A4: Receiving IPv6 Packet with Errors

This test purpose presents the system that will act differently according to the value of the first two bits in unrecognized option type error. In figure A4.1, the first transition shows the system receiving an IPv6 packet from link layer. This packet contains errors in destination header2 (the option type '10000001' is unrecognized), consequently, this packet is discard and an ICMP error message is sent, which is shown in the second transition. The third transition shows the system receiving another IPv6 packet, which contains errors in destination header2 (the option type '00000011' is unrecognized), as a result, this error is skipped, and the PDU is sent out to the transport layer. Figure A4.2 is the corresponding test case.

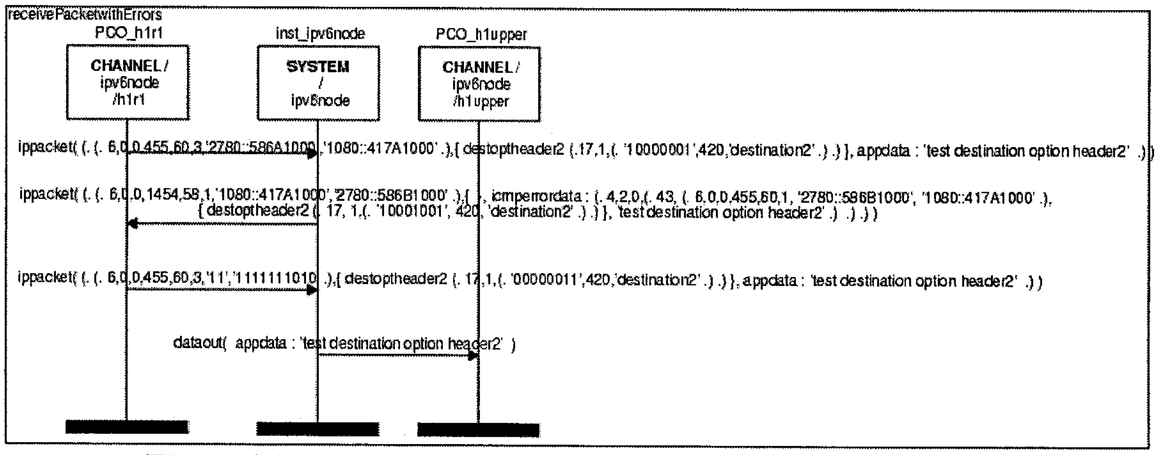

receivePacketwithErrors
PCO_h1r1
inst_ipv6node
PCO_h1upper
CHANNEL / ipv6node /h1r1
SYSTEM / ipv6node
CHANNEL / ipv6node /h1upper
ippacket((. (. 6,0,0,455,60,3,'2780::586A1000','1080::417A1000' .),{ destoptheader2 (.17,1,(. '10000001',420,'destination2' .) .) }, appdata : 'test destination option header2' .))
ippacket((. (. 6,0,0,1454,58,1,'1080::417A1000','2780::586B1000' .),{ }, icmperrordata : (. 4,2,0,(. 43, (. 6,0,0,455,60,1, '2780::586B1000', '1080::417A1000' .), { destoptheader2 (. 17, 1,(. '10001001', 420, 'destination2' .) .) }, 'test destination option header2' .) .) .))
ippacket((. (. 6,0,0,455,60,3,'11','1111111010' .),{ destoptheader2 (. 17,1,(. '00000011',420,'destination2' .) .) }, appdata : 'test destination option header2' .))
dataout(appdata : 'test destination option header2')

Figure A4.1 test purpose for receiving IPv6 packets with errors

IPv6Node May 2, 2004 ITEX 4.1.1

Test Case Dynamic Behaviour

Test Case Name : receivePacketwithErrors
Group :
Purpose : from state idle of ipv6host, receive ippacket, send (ippacket) and go to state idle
Configuration :
Default : DEF_0
Comments : Generated by test oriented simulation of the current test purpose scenario

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		h1r1!ippacket START TAC	ippacket_11		
2		h1r1?ippacket CANCEL TAC	ippacket_12		
3		h1r1!ippacket START TAC	ippacket_13		
4		h1upper?dataout CANCEL TAC	dataout_14	(P)	

Detailed Comments :

Figure A4.2 test case for receiving IPv6 packets with errors

A5: Sending and Receiving ICMP Informational Messages

This test purpose describes the basic behaviors of a host in ICMP protocol. In figure A5.1, the first transition is the request from transport layer. Therefore, an Echo Request is sent. It is shown in the second transition. The third transition shows that a destination receives an Echo Reply. As a result, the destination passes the Echo Reply message to the transport layer, which is shown in the fourth transition. The fifth transition shows a destination receiving an Echo request. Therefore, an Echo Reply is sent. It is shown in the last transition. Figure A5.2 is the corresponding teat case.

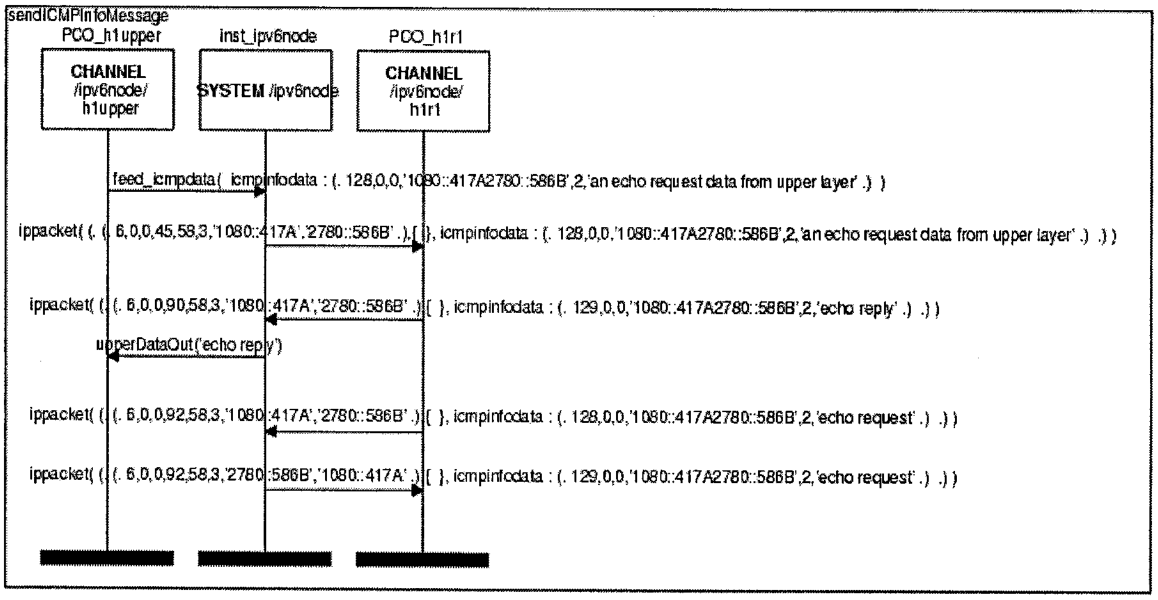
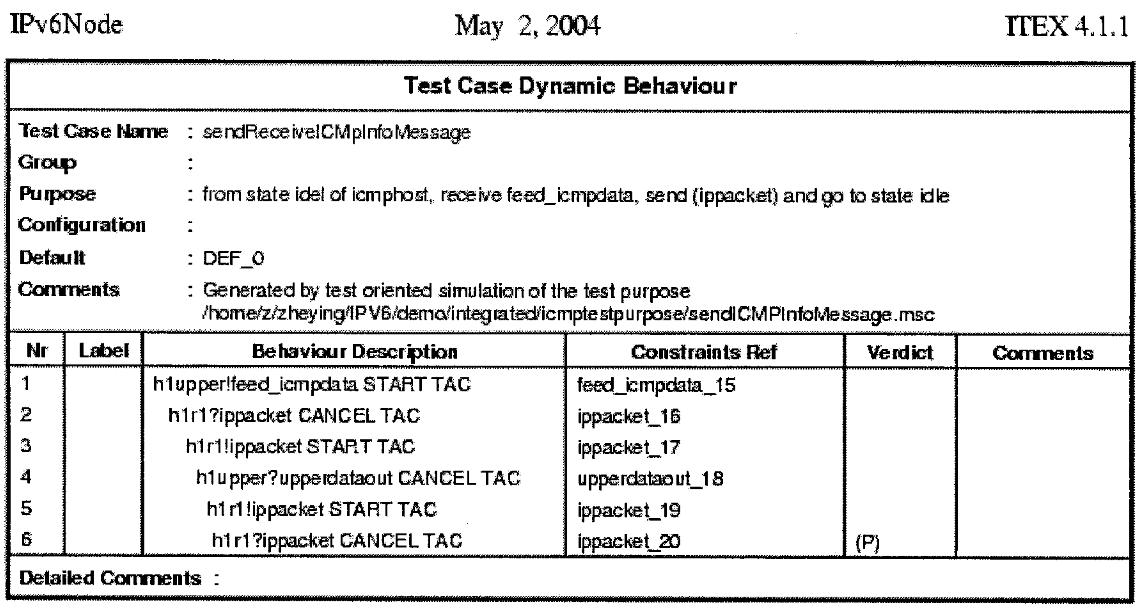

sendICMPInfoMessage
PCO_h1upper
inst_ipv6node
PCO_h1r1
CHANNEL /ipv6node/ h1upper
SYSTEM /ipv6node
CHANNEL /ipv6node/ h1r1
feed_icmpdata(icmpinfodata : (. 128,0,0,'1080::417A2780::586B',2,'an echo request data from upper layer' .))
ippacket((. (. 6,0,0,45,58,3,'1080::417A','2780::586B' .),{ }, icmpinfodata : (. 128,0,0,'1080::417A2780::586B',2,'an echo request data from upper layer' .) .))
ippacket(((. 6,0,0,90,58,3,'1080::417A','2780::586B' .)[}, icmpinfodata : (. 129,0,0,'1080::417A2780::586B',2,'echo reply' .) .))
upperDataOut('echo reply')
ippacket(((. 6,0,0,92,58,3,'1080::417A','2780::586B' .)[}, icmpinfodata : (. 128,0,0,'1080::417A2780::586B',2,'echo request' .) .))
ippacket(((. 6,0,0,92,58,3,'2780::586B','1080::417A' .)[}, icmpinfodata : (. 129,0,0,'1080::417A2780::586B',2,'echo request' .) .))

Figure A5.1 test purpose for sending and receiving ICMP informational messages



IPv6Node May 2, 2004 ITEX 4.1.1

Test Case Dynamic Behaviour	
Test Case Name	: sendReceiveICMPInfoMessage
Group	:
Purpose	: from state idel of icmphost, receive feed_icmpdata, send (ippacket) and go to state idle
Configuration	:
Default	: DEF_0
Comments	: Generated by test oriented simulation of the test purpose /home/z/zheying/IPV6/demo/integrated/icmptestpurpose/sendICMPInfoMessage.msc

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		h1upper!feed_icmpdata START TAC	feed_icmpdata_15		
2		h1r1?ippacket CANCEL TAC	ippacket_16		
3		h1r1!ippacket START TAC	ippacket_17		
4		h1upper?upperdataout CANCEL TAC	upperdataout_18		
5		h1r1!ippacket START TAC	ippacket_19		
6		h1r1?ippacket CANCEL TAC	ippacket_20	(P)	

Detailed Comments :

Figure A5.2 test case for sending and receiving ICMP informational messages

A6: Sending and Receiving MLD Messages

This test purpose shows that when a host is interested in a multicast address, the host sends unsolicited report. In figure A6.1, the first transition shows that the unsolicited report timer times out. As a result, listener reports are sent, which are shown in the second and third transitions. The last transition shows the host receiving a General Query, which causes the host to reset Maximum Query Response timer internally. Figure A6.2 is the corresponding test case.

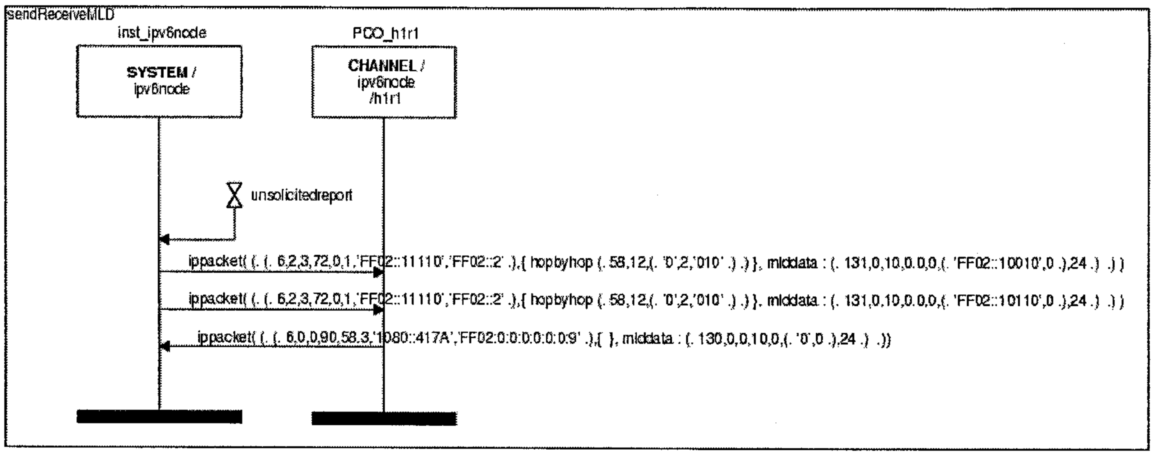
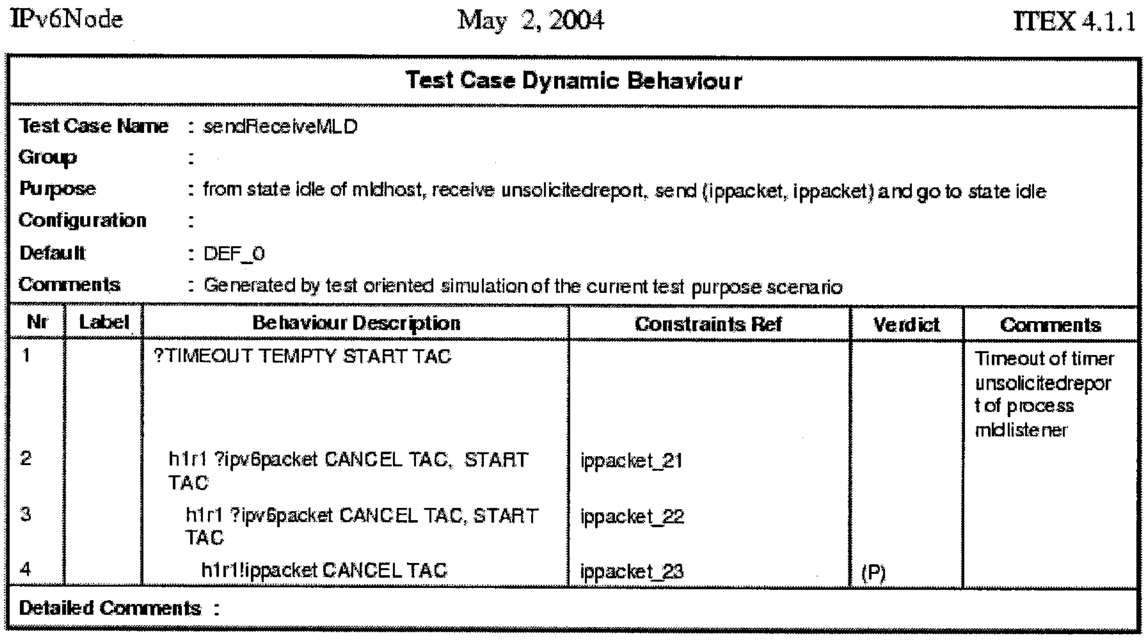

sendReceiveMLD
inst_ipv6node
PCO_h1r1
SYSTEM / ipv6node
CHANNEL / ipv6node /h1r1
unsolicitedreport
ippacket((. (. 6,2,3,72,0,1,'FF02::11110','FF02::2' .),{ hopbyhop (. 58,12,(. '0',2,'010' .) .) }, mlddata : (. 131,0,10,0.0,0,(. 'FF02::10010',0 .),24 .) .))
ippacket((. (. 6,2,3,72,0,1,'FF02::11110','FF02::2' .),{ hopbyhop (. 58,12,(. '0',2,'010' .) .) }, mlddata : (. 131,0,10,0.0,0,(. 'FF02::10110',0 .),24 .) .))
ippacket((. (. 6,0,0,90,58,3,'1080::417A','FF02:0:0:0:0:0:0:9' .),{ }, mlddata : (. 130,0,0,10,0,(. '0',0 .),24 .) .))

Figure A6.1 test purpose for sending and receiving MLD messages



IPv6Node May 2, 2004 ITEX 4.1.1

Test Case Dynamic Behaviour

Test Case Name : sendReceiveMLD
Group :
Purpose : from state idle of mldhost, receive unsolicitedreport, send (ippacket, ippacket) and go to state idle
Configuration :
Default : DEF_0
Comments : Generated by test oriented simulation of the current test purpose scenario

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		?TIMEOUT TEMPTY START TAC			Timeout of timer unsolicitedrepor t of process mldlistener
2		h1r1 ?ipv6packet CANCEL TAC, START TAC	ippacket_21		
3		h1r1 ?ipv6packet CANCEL TAC, START TAC	ippacket_22		
4		h1r1!ippacket CANCEL TAC	ippacket_23	(P)	

Detailed Comments :

Figure A6.2 test case for sending and receiving MLD messages

A7: Maximum Query Response Timeout

This test purpose shows that a host will send [Startup Count] Listener Reports after the Maximum Query Response Timer timeout. Figure A7.2 is the corresponding test case.

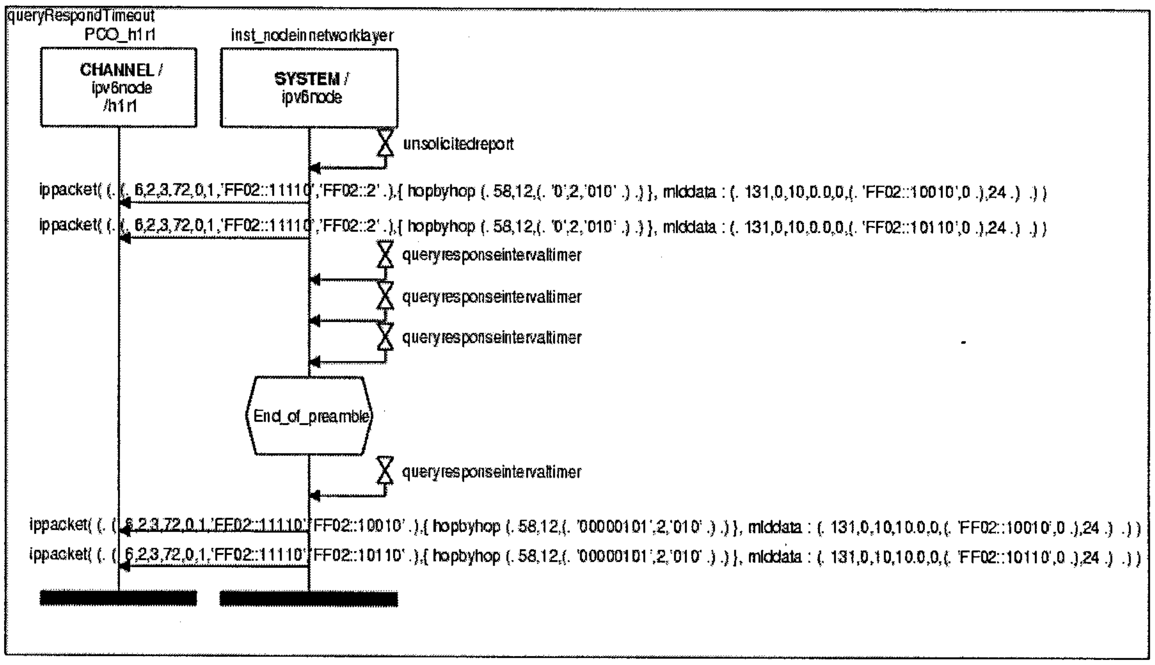

queryRespondTimeout
PCO_h1r1
inst_nodeinnetworklayer
CHANNEL / ipv6node /h1r1
SYSTEM / ipv6node
unsolicitedreport
ippacket((. (. 6,2,3,72,0,1,'FF02::11110','FF02::2' .),{ hopbyhop (. 58,12,(. '0',2,'010' .) .) }, mlddata : (. 131,0,10,0.0,0,(. 'FF02::10010',0 .),24 .) .))
ippacket((. (. 6,2,3,72,0,1,'FF02::11110','FF02::2' .),{ hopbyhop (. 58,12,(. '0',2,'010' .) .) }, mlddata : (. 131,0,10,0.0,0,(. 'FF02::10110',0 .),24 .) .))
queryresponseintervaltimer
queryresponseintervaltimer
queryresponseintervaltimer
End_of_preamble
queryresponseintervaltimer
ippacket((. (. 6,2,3,72,0,1,'FF02::11110','FF02::10010' .),{ hopbyhop (. 58,12,(. '00000101',2,'010' .) .) }, mlddata : (. 131,0,10,10.0,0,(. 'FF02::10010',0 .),24 .) .))
ippacket((. (. 6,2,3,72,0,1,'FF02::11110','FF02::10110' .),{ hopbyhop (. 58,12,(. '00000101',2,'010' .) .) }, mlddata : (. 131,0,10,10.0,0,(. 'FF02::10110',0 .),24 .) .))

Figure A7.1 test purpose for maximum query response timeout

Test Case Dynamic Behaviour

Test Case Name	: queryRespondTimeout
Group	:
Purpose	: from state idle of mldhost, receive queryresponseintervaltimer, send (ipv6packet, ipv6packet) and go to state idle
Configuration	:
Default	: DEF_0
Comments	: Generated by test oriented simulation of the current test purpose scenario

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		?TIMEOUT TEMPTY START TAC			Timeout of timer unsolicitedreport of process mldlistener
2		h1r1 ?ippacket CANCEL TAC, START TAC	ippacket_21		
3		h1r1 ?ippacket CANCEL TAC, START TEMPTY(10)	ippacket_22		
4		?TIMEOUT TEMPTY START TEMPTY(20)			Timeout of timer queryresponseintervaltimer of process mldlistener
5		?TIMEOUT TEMPTY START TEMPTY(20)			Timeout of timer queryresponseintervaltimer of process mldlistener
6		?TIMEOUT TEMPTY START TEMPTY(20)			Timeout of timer queryresponseintervaltimer of process mldlistener
7		?TIMEOUT TEMPTY START TAC			Timeout of timer queryresponseintervaltimer of process mldlistener
8		h1r1 ?ippacket CANCEL TAC, START TAC	ippacket_23		
9		h1r1 ?ippacket CANCEL TAC	ippacket_24	(P)	

Detailed Comments :

Figure A7.2 test case for maximum query response timeout

Appendix B: Conformance Test Cases for Routers

This section presents selected test purposes and their corresponding test cases for conformance testing for IPv6 routers. An IPv6 router considered here should have properties of IPv6, ICMPv6, RIPng and MLD protocols. The selected test purposes should have proper input data, whose observable behaviours should be shown after it stimulates the SUT. The following are these selected test purposes and corresponding test cases.

B1: Basic Behaviors of Routers in IPv6 and RIPng

This test purpose presents the basic behaviors of a router in IPv6 and RIPng protocols. In figure B1.1, the first transition shows the router sending a General Query in order to set up its Querier state. The second transition shows the router receiving an IPv6 packet, which its destination is the address of this router. In other word, the router is a destination router for this packet. After the router analyzes the received packet, it forwards this packet to a host, which connects with the router directly. The following two transitions have the same explanation. The difference is that the received packet is forwarded to another host. The last two transitions show that the router is an intermediate router because the received packet is forwarded to another router. Figure B1.2 is the corresponding test case.

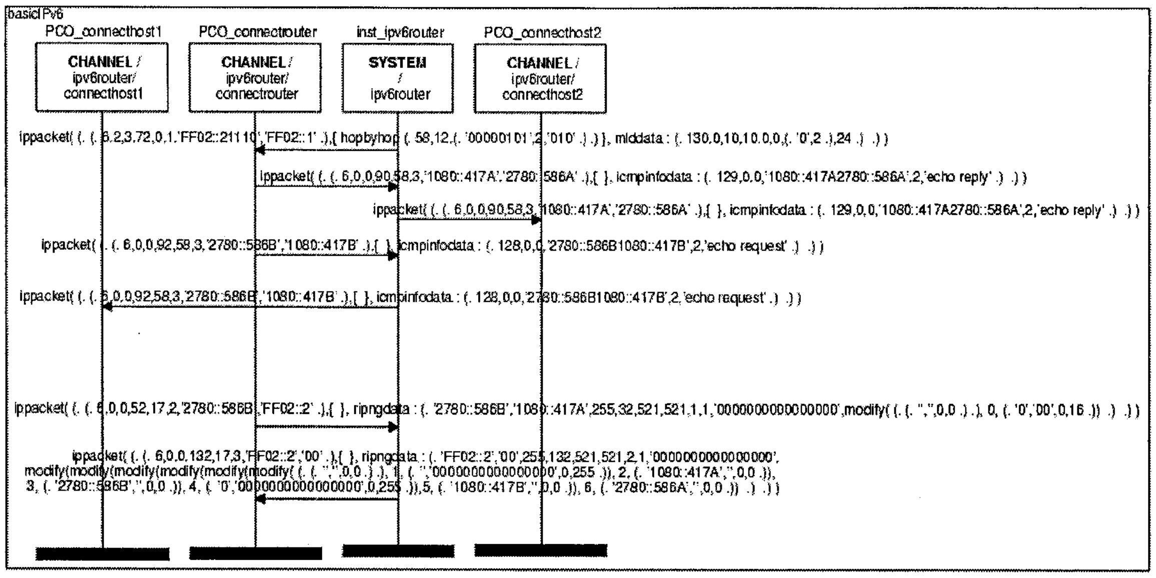
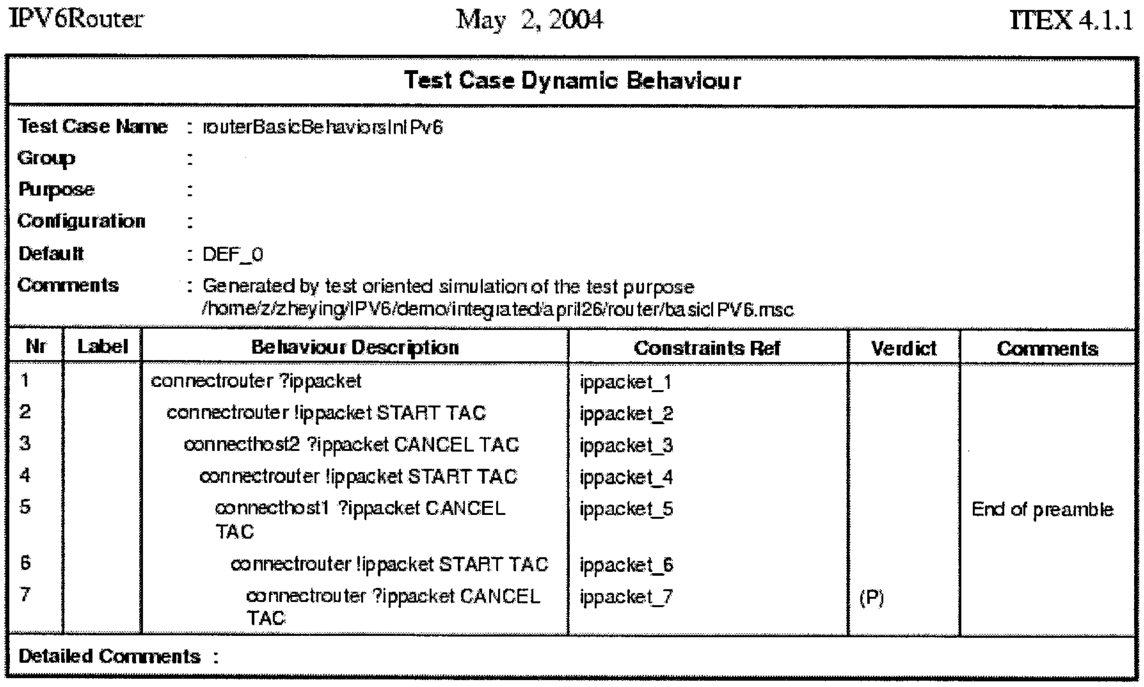

basicIPv6
PCO_connecthost1
PCO_connectrouter
inst_ipv6router
PCO_connecthost2
CHANNEL / ipv6router/ connecthost1
CHANNEL / ipv6router/ connectrouter
SYSTEM / ipv6router
CHANNEL / ipv6router/ connecthost2
ippacket((. (. 6,2,3,72,0,1,'FF02::21110','FF02::1' .),{ hopbyhop (. 58,12,(. '00000101',2,'010' .) .) }, mlddata : (. 130,0,10,10,0,0,(. '0',2 .),24 .) .))
ippacket((. (. 6,0,0,90,58,3,'1080::417A','2780::586A' .),{ }, icmpinfodata : (. 129,0,0,'1080::417A2780::586A',2,'echo reply' .) .))
ippacket((. (. 6,0,0,90,58,3,'1080::417A','2780::586A' .),{ }, icmpinfodata : (. 129,0,0,'1080::417A2780::586A',2,'echo reply' .) .))
ippacket((. (. 6,0,0,92,58,3,'2780::586B','1080::417B' .),{ }, icmpinfodata : (. 128,0,0,'2780::586B1080::417B',2,'echo request' .) .))
ippacket((. (. 6,0,0,92,58,3,'2780::586B','1080::417B' .),{ }, icmpinfodata : (. 128,0,0,'2780::586B1080::417B',2,'echo request' .) .))
ippacket((. (. 6,0,0,52,17,2,'2780::586B','FF02::2' .),{ }, ripngdata : (. '2780::586B','1080::417A',255,32,521,521,1,1,'0000000000000000',modify((. (. '','',0,0 .) .), 0, (. '0','00',0,16 .)) .) .))
ippacket((. (. 6,0,0,132,17,3,'FF02::2','00' .),{ }, ripngdata : (. 'FF02::2','00',255,132,521,521,2,1,'0000000000000000', modify(modify(modify(modify(modify(modify((. (. '','',0,0 .) .), 1, (. '','0000000000000000',0,255 .)), 2, (. '1080::417A','',0,0 .)), 3, (. '2780::586B','',0,0 .)), 4, (. '0','0000000000000000',0,255 .)),5, (. '1080::417B','',0,0 .)), 6, (. '2780::586A','',0,0 .)) .) .))

Figure B1.1 test purpose for basic behaviors of routers in IPv6 and RIPng


IPV6Router May 2, 2004 ITEX 4.1.1

Test Case Dynamic Behaviour

Test Case Name	:	routerBasicBehaviorsInIPv6
Group	:	
Purpose	:	
Configuration	:	
Default	:	DEF_0
Comments	:	Generated by test oriented simulation of the test purpose /home/z/zheying/IPV6/demo/integrated/april26/router/basicIPV6.msc

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		connectrouter ?ippacket	ippacket_1		
2		connectrouter !ippacket START TAC	ippacket_2		
3		connecthost2 ?ippacket CANCEL TAC	ippacket_3		
4		connectrouter !ippacket START TAC	ippacket_4		
5		connecthost1 ?ippacket CANCEL TAC	ippacket_5		End of preamble
6		connectrouter !ippacket START TAC	ippacket_6		
7		connectrouter ?ippacket CANCEL TAC	ippacket_7	(P)	

Detailed Comments :

Figure B1.2 test case for basic behaviors of routers in IPv6 and RIPng

B2: Basic Behaviors of Routers in ICMP

This test purpose presents the basic behaviors of a router in ICMP protocol. In figure B2.1, the first transition is a General Query, which is for setting the Querier state of the router. The second transition is the request from transport layer. Therefore, an Echo Request is sent. This is shown in the following transition. The fourth transition shows that a destination receives an Echo Reply. As a result, the destination passes the Echo Reply message to the transport layer, which is shown in the following transition. The sixth transition shows a destination receiving an Echo request. Therefore, an Echo Reply is sent. It is shown in the last transition. Figure B2.2 is the corresponding test case.

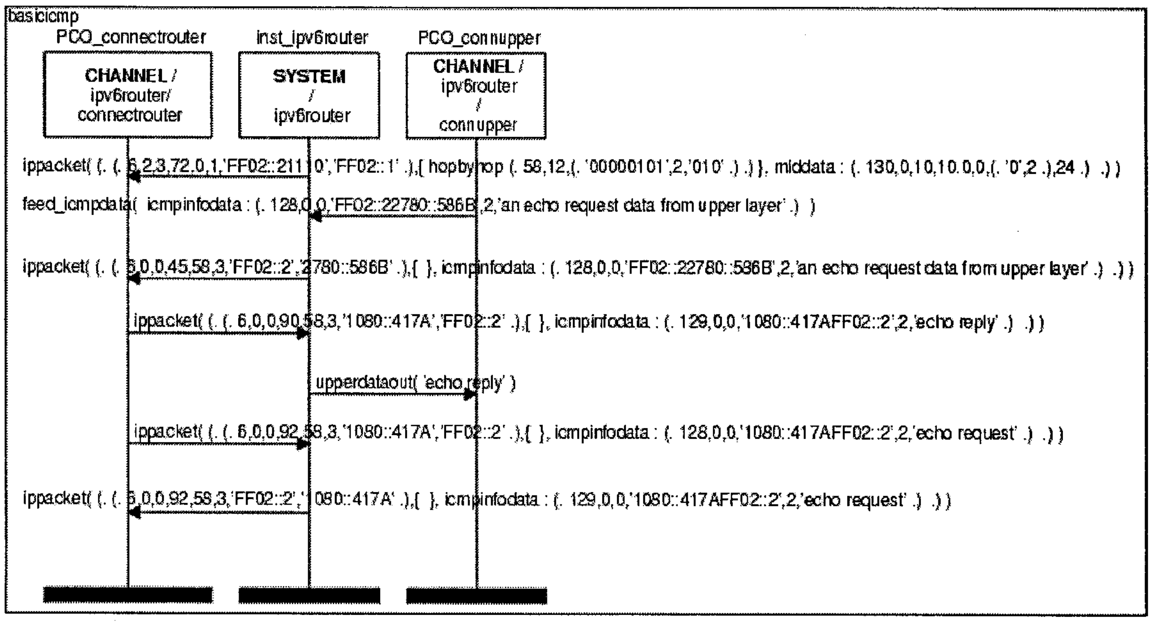

basicicmp
PCO_connectrouter
inst_ipv6router
PCO_connupper
CHANNEL / ipv6router/ connectrouter
SYSTEM / ipv6router
CHANNEL / ipv6router / connupper
ippacket((. (. 6,2,3,72,0,1,'FF02::2110','FF02::1' .),{ hopbyhop (. 58,12,(. '00000101',2,'010' .) .) }, mlddata : (. 130,0,10,10.0,0,(. '0',2 .),24 .) .))
feed_icmpdata(icmpinfodata : (. 128,0,0,'FF02::22780::586B',2,'an echo request data from upper layer' .))
ippacket((. (. 6,0,0,45,58,3,'FF02::2','2780::586B' .),{ }, icmpinfodata : (. 128,0,0,'FF02::22780::586B',2,'an echo request data from upper layer' .) .))
ippacket((. (. 6,0,0,90,58,3,'1080::417A','FF02::2' .),{ }, icmpinfodata : (. 129,0,0,'1080::417AFF02::2',2,'echo reply' .) .))
upperdataout('echo reply')
ippacket((. (. 6,0,0,92,58,3,'1080::417A','FF02::2' .),{ }, icmpinfodata : (. 128,0,0,'1080::417AFF02::2',2,'echo request' .) .))
ippacket((. (. 6,0,0,92,58,3,'FF02::2','1080::417A' .),{ }, icmpinfodata : (. 129,0,0,'1080::417AFF02::2',2,'echo request' .) .))

Figure B2.1 test purpose for basic behaviors of routers in ICMP

IPV6Router May 2, 2004 ITEX 4.1.1

Test Case Dynamic Behaviour

Test Case Name	: routerBasicBehaviorsInICMP
Group	:
Purpose	:
Configuration	:
Default	: DEF_0
Comments	: Generated by test oriented simulation of the test purpose /home/z/zheying/IPV6/demo/integrated/april26/router/basicICMP.msc

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		connectrouter ?ippacket	ippacket_1		
2		connupper !feed_icmpdata START TAC	feed_icmpdata_2		
3		connectrouter ?ippacket CANCEL TAC	ippacket_3		
4		connectrouter !ippacket START TAC	ippacket_4		
5		connupper ?upperdataout CANCEL TAC	upperdataout_5		End of preamble
6		connectrouter !ippacket START TAC	ippacket_6		
7		connectrouter ?ippacket CANCEL TAC	ippacket_7	(P)	

Detailed Comments :

Figure B2.2 test case for basic behaviors of routers in ICMP

B3: Basic Behaviors of Routers in MLD

This test purpose presents the basic behaviors of a router in MLD. In figure B3.1, the first two transitions send two General Query at startup of the system. After General Query Interval Timer timeout, the router sends two General Query again; these General Query are the same General queries sent at startup. The second to last transition shows that the router receives a Listener Report message and then adds the multicast address it contains into the multicast address list internally. The last transition shows that the router receives a Listener Done message and then deletes the multicast address it contains from the multicast address list internally. Figure B3.2 is the corresponding test case.

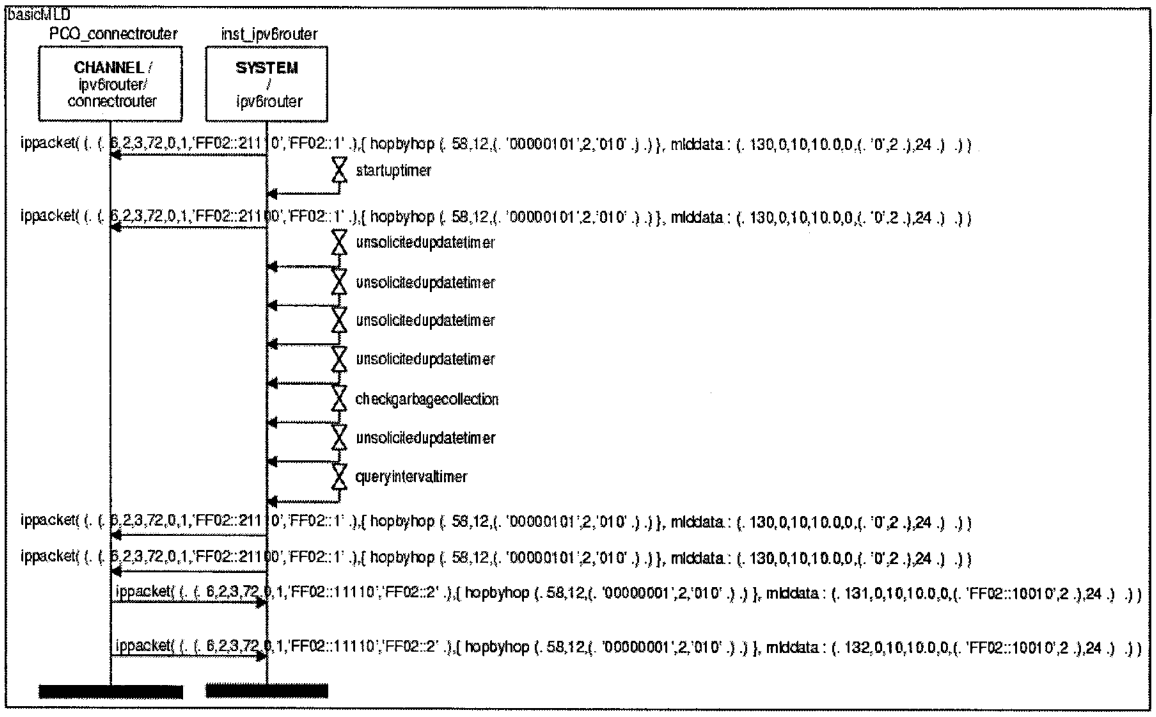

basicMLD
PCO_connectrouter
inst_ipv6router
CHANNEL / ipv6router/ connectrouter
SYSTEM / ipv6router
ippacket((. (. 6,2,3,72,0,1,'FF02::2110','FF02::1' .),{ hopbyhop (. 58,12,(. '00000101',2,'010' .) .) }, mlddata : (. 130,0,10,10.0.0,(. '0',2 .),24 .) .))
startuptimer
ippacket((. (. 6,2,3,72,0,1,'FF02::2110','FF02::1' .),{ hopbyhop (. 58,12,(. '00000101',2,'010' .) .) }, mlddata : (. 130,0,10,10.0.0,(. '0',2 .),24 .) .))
unsolicitedupdatetimer
unsolicitedupdatetimer
unsolicitedupdatetimer
unsolicitedupdatetimer
checkgarbagecollection
unsolicitedupdatetimer
queryintervaltimer
ippacket((. (. 6,2,3,72,0,1,'FF02::2110','FF02::1' .),{ hopbyhop (. 58,12,(. '00000101',2,'010' .) .) }, mlddata : (. 130,0,10,10.0.0,(. '0',2 .),24 .) .))
ippacket((. (. 6,2,3,72,0,1,'FF02::2110','FF02::1' .),{ hopbyhop (. 58,12,(. '00000101',2,'010' .) .) }, mlddata : (. 130,0,10,10.0.0,(. '0',2 .),24 .) .))
ippacket((. (. 6,2,3,72,0,1,'FF02::11110','FF02::2' .),{ hopbyhop (. 58,12,(. '00000001',2,'010' .) .) }, mlddata : (. 131,0,10,10.0.0,(. 'FF02::10010',2 .),24 .) .))
ippacket((. (. 6,2,3,72,0,1,'FF02::11110','FF02::2' .),{ hopbyhop (. 58,12,(. '00000001',2,'010' .) .) }, mlddata : (. 132,0,10,10.0.0,(. 'FF02::10010',2 .),24 .) .))

Figure B3.1 test purpose for basic behaviors of routers in MLD

Test Case Dynamic Behaviour

Test Case Name	: routerBasicBehaviorsInMLD
Group	:
Purpose	:
Configuration	:
Default	: DEF_0
Comments	: Generated by test oriented simulation of the test purpose /home/z/zheying/IPV6/demo/integrated/april26/router/basicMLD.msc

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		connectrouter ?ippacket START TEMPTY(30)	ippacket_1		
2		?TIMEOUT TEMPTY START TAC			Timeout of timer startuptimer of process mdlquerier
3		connectrouter ?ippacket CANCEL TAC, START TEMPTY	ippacket_2		
4		?TIMEOUT TEMPTY START TEMPTY(30)			Timeout of timer unsolicitedupdatetimer of process ripng
5		?TIMEOUT TEMPTY START TEMPTY(30)			Timeout of timer unsolicitedupdatetimer of process ripng
6		?TIMEOUT TEMPTY START TEMPTY(30)			Timeout of timer unsolicitedupdatetimer of process ripng
7		?TIMEOUT TEMPTY START TEMPTY			Timeout of timer checkgarbagecollection of process ripng
8		?TIMEOUT TEMPTY START TEMPTY(30)			Timeout of timer unsolicitedupdatetimer of process ripng
9		?TIMEOUT TEMPTY START TEMPTY(5)			Timeout of timer unsolicitedupdatetimer of process ripng
10		?TIMEOUT TEMPTY START TAC			Timeout of timer queryintervaltimer of process mdlquerier
11		connectrouter ?ippacket CANCEL TAC, START TAC	ippacket_1		
12		connectrouter ?ippacket CANCEL TAC	ippacket_2		
13		connectrouter !ippacket	ippacket_3		End of preamble
14		connectrouter !ippacket	ippacket_4	(P)	

Detailed Comments :

Figure B3.2 test case for basic behaviors of routers in MLD