

FORMAL SPECIFICATION OF FRAGMENTATION AND REASSEMBLY IN IPv6

by

İbrahim Şahin

B.Sc. June 1993, Gazi University

A Thesis Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

ELECTRICAL ENGINEERING

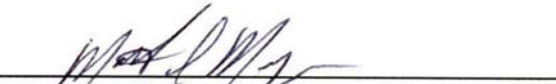
OLD DOMINION UNIVERSITY

December 1997

Approved by:



James F. Leathrum, Jr.



Martin D. Meyer



Oscar R. González

ABSTRACT

FORMAL SPECIFICATION OF FRAGMENTATION AND REASSEMBLY IN IPv6

İbrahim Şahin
Old Dominion University, 1997
Director: Dr. James F. Leathrum, Jr.

Development and implementation of a networking standard such as the new Internet Protocol (IPv6) is a very difficult process. Different implementations of the standard must be fully compatible to allow different computers to communicate with each other. However, standards are often ambiguous, frequently a result of providing specifications in the English language. A more formal specification could assist in the design of systems. This thesis demonstrates that capability using the Prototype Verification System (PVS).

In this thesis study, a formal specification for fragmentation and reassembly in IPv6 was created to provide a tool for the standardization of IPv6 using PVS. The user of the specification can employ it in verifying the design of his implementation. The user can pass his function implementations to the specification as parameters. The specification requires the user to provide certain properties of his functions.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Dr. James F. Leathrum, Jr., for his invaluable guidance and his encouragement through the course of this research. This thesis would not have been possible without his knowledge, wisdom, and direction.

I would also like to thank the additional members of the thesis advisory committee, Martin D. Meyer, and Oscar R. González for their suggestions, comments, and beneficial discussions.

I would like to extend my thanks to Paul S. Miner for his assistance with Prototype Verification Systems.

In addition, I would like to thank all my coworkers in the Formal Methods group, particularly my friends Hüseyin Özgüngör and Rasha M. B. E. Morsi for their help with knowledge and suggestions.

Finally, I would like to thank Zeki B. Hamşioğlu for his great help with the writing of this thesis in L^AT_EX.

TABLE OF CONTENTS

	<u>Page</u>
ABSTRACT	
ACKNOWLEDGMENTS	iii
LIST OF FIGURES	vii
I INTRODUCTION	1
1.1 Background	1
1.1.1 Internet Protocol (IP)	1
1.1.2 Formal Methods	3
1.2 Objectives	4
1.3 Outline of the Thesis	6
II THE NEW INTERNET PROTOCOL (IPv6)	7
2.1 IPv6: The New Internet Protocol	7
2.1.1 IPv6 Header Format	12
2.1.2 Fragmentation and Reassembly	18
2.1.2.1 MTU and Path MTU	20
2.1.2.2 Fragmentation and Reassembly in IPv4	22
2.1.2.3 Fragmentation and Reassembly in IPv6	22
III FORMAL METHODS IN COMMUNICATION PROTOCOLS	30
3.1 Formal Methods and PVS	30
3.1.1 Formal Methods	30
3.1.2 PVS: Prototype Verification System	32
3.1.2.1 The PVS Specification Language	34
3.1.2.2 The PVS Proof Checker	35

3.1.3	Formal Specification and Verification Studies	38
3.2	The Header Specification	40
IV SPECIFICATION OF THE FRAGMENTATION AND		
REASSEMBLY		45
4.1	Approach to the Fragmentation and Reassembly	45
4.2	PVS Specification	48
4.2.1	Type Specifications	49
4.2.2	Fragmentation	52
4.2.3	Reassembly	59
4.2.4	The <code>transmission()</code> Function.	65
V VERIFICATION OF THE FRAGMENTATION AND		
REASSEMBLY PROCESS		66
5.1	Lemmas for Verification	66
5.2	The <code>result</code> Theorem	71
5.3	Typechecking the Specification	72
5.4	Proving the Lemmas and the Theorem	74
VI CONCLUSIONS		84
6.1	Results	84
6.2	Future Research	85
REFERENCES		87
APPENDICES: THE SPECIFICATION AND PROOF FILES		92
A	The Specification Files	92
A.1	Specification of the IPv6 Header	92
A.2	Type Specifications	95
A.3	Specification of the Fragmentation and Reassembly	96

B	The Proof Files	105
B.1	The Proof File of Type Specifications	105
B.2	The Proof File of Fragmentation and Reassembly	105
VITA		108

LIST OF FIGURES

	<u>Page</u>
2.1 The layers of the TCP/IP protocols.	8
2.2 Flow of data through the layers of the TCP/IP protocols.	9
2.3 The number of hosts on the Internet by year [8].	11
2.4 The IPv6 packet with all the extension headers.	13
2.5 The IPv6 and IPv4 header formats.	14
2.6 The IPv6 extension header formats.	16
2.7 The unfragmented original packet.	24
2.8 The original packet after dividing into fragments.	25
2.9 The fragment packets.	26
2.10 The IPv6 fragmentation example.	27
4.1 Data flow for the fragmentation process.	46
4.2 Data flow for the reassembly process.	48

CHAPTER I

INTRODUCTION

In this thesis, fragmentation and reassembly in the new Internet Protocol (IPv6) are formally specified and verified in conjunction with the work in [40]. There are two purposes for formally specifying the fragmentation and reassembly in IPv6. The first one is to create a tool for the standardization process of IPv6 to provide an unambiguous standard. Generally, protocol specifications are written in the English language, resulting in the possibility of including ambiguity. The second purpose is to allow implementers to reason about their design preferences and validate their design choices against an unambiguous standard. This work used the Prototype Verification System (PVS), a formal specification and verification environment, to specify and verify certain properties of the fragmentation and reassembly process of the new Internet Protocol.

1.1 Background

1.1.1 Internet Protocol (IP)

The Internet can be defined as a set of networks including local networks at a number of institutions and a number of military networks. It is created by connecting hosts with some data routing devices and physical links. To make data communication possible, some standard communication protocols such as Transmission Control Protocol (TCP) and User Datagram Protocol (UDP), must be implemented at each host or at each data routing device.

Journal model used for this thesis is *IEEE Communication Magazine*.

The Transmission Control Protocol/Internet Protocol (TCP/IP) is a member of the major data communication protocol family which makes data communication between computers possible over the Internet [2]. The Internet Protocol (IP) is one of the TCP/IP protocol family members. Its duty is to transmit all upper layer protocols such as TCP, UDP data and Internet Control Message Protocol (ICMP) data across the network.

Currently, in most TCP/IP implementations, Internet Protocol version 4 (IPv4) is being used as a network layer protocol [7]. As the Internet grows, IPv4 has become insufficient to meet the performance and functional requirements for the Internet [19]. There are two major reasons that IPv4 has become inadequate. The first reason is that IPv4 is running out of network addresses due to the increasing popularity of the Internet. The second reason is the explosion in the size of routing tables [6]. Due to the inabilities of IPv4, the Internet Engineering Task Force (IETF) formed a group to evaluate a new Internet Protocol. The new Internet Protocol (IPv6) recommendation was approved by IETF in November 1994 and the specification was introduced in the Request For Comments (RFC) 1883 [15].

Changing the Internet Protocol is more than simply changing a single protocol. Changes to IP affect at least 58 current TCP/IP standards [6, 9]. A change in the protocol means reimplementing of the protocol for different types of computers, because while each computer has a different structure, it should have the same protocol to communicate with other computers.

Implementing a protocol is very time consuming and costly. It is almost impossible to design an error free implementation. An error at the design stage of an implementation may cause a redesign of the implementation from the beginning. For that reason, a manufacturer needs to be sure that his implementation will

work. This requirement of the manufacturer can be met by formally specifying and verifying the protocol and his design using formal methods.

1.1.2 Formal Methods

Formal methods refers to the application of mathematical techniques for the specification, analysis, and design of complex computer software and hardware systems [27]. Using a formal notation increases the understanding of the operation of a system, especially early in the design stage. Such methods can assist the design team in thinking about the operation of the system before it is implemented.

Formal methods can be used to be sure that an initial design is sound before implementing the design. Before implementing in software, the program structure can be specified in a formal technique. If the specification of the implementation is verified successfully by using formal methods, there is a higher confidence in the correctness of the functionality.

Today, a number of formal specification environments are available [28]. The Prototype Verification System (PVS) is one of these specification environments. It was developed at SRI International Computer Science Laboratory. PVS is an integrated environment for constructing and analyzing clear and precise formal specifications. It supports a variety of facilities such as creating, analyzing and documenting theories and proofs [32, 33].

Formal methods have been used in the design of a great number of systems. Leathrum et al. specified and verified parts of IPv6 in PVS. They developed theories for networks, IPv6 headers (including the basic IPv6 header and all extension header) and IPv6 packets, and verified some properties of routing in IPv6. Their work provides the framework upon which this thesis is built.

1.2 Objectives

To build a complete specification of IPv6, fragmentation and reassembly must be specified. In this research, this part of the new Internet Protocol is formally specified. The main objectives of this thesis can be listed as follows:

- **Building a formal specification of fragmentation and reassembly:**

Most of the time, after a proposal of a communication protocol is approved, the specification of the protocol is written in the English language. Although they are written very carefully, sometimes there is ambiguity, thus they may be understood differently by different people. On the other hand, a mathematical definition expresses the idea precisely. For that reason, the fragmentation and reassembly process is specified formally using a mathematical definition method.

- **Verification:** A specification written in the English language cannot be verified practically, whereas it is possible to verify a formal specification by using some proof checker tools. For that reason, when it is required to verify a specification written in the English language, it is first specified in a formal language. Simply specifying a system or protocol formally does not mean that it is correct. There could be some errors in the original specification or even in the formal specification done during the conversion. Thus, the formal specification must be verified before both original and formal specifications are considered correct. Verifying the formal specification of fragmentation and reassembly process is another objective. After this process is specified formally, its certain properties are verified by using PVS proof checker. By verifying it, we show that certain properties of both the original and the formal specifications are correct.

- **Assisting implementers:** A new communication protocol is usually implemented by different implementers for different kinds of computers. To make communication possible each implementation must work exactly in the same way. This depends on the implementers' understanding of the specification. Implementers can use a formal specification rather than a specification written in the English language. In this study, a formal specification for fragmentation and reassembly process of the new IP is created to help implementers understand the specification. Implementers also can use the specification in verifying their design choices before they implement their design.
- **Creating a formal tool to assist in the design of implementations:** Another objective is that by formally specifying the fragmentation and reassembly process to create a formal environment for the standardization process of IPv6 to provide an unambiguous standard. The user of the environment can reason about his design preferences. By using the environment, the user can see whether or not his design meets certain requirements of the original specification. The user can pass his function implementations to the specification as parameters. The specification requires the user to provide certain properties of his functions.

For the specification, a functional approach is taken. Several functions are specified to define the fragmentation and reassembly process. Functions are specified as parameters to the specification so that the user of the specification can pass his implementation of these functions to the specification. In the specification, for each function, assumptions about its behavior are specified. Because this is a formal specification, functions and their assumptions are defined behaviorally instead of structurally.

1.3 Outline of the Thesis

The presentation of the research is divided into number of chapters. In Chapter I, the thesis and the objectives of this research are introduced. The outline of the thesis is also provided in Chapter I. Background information about the new Internet Protocol is given in Chapter II. In Chapter III, formal methods and PVS are introduced. Some formal method studies and the header specification of IPv6 are also explained in Chapter III. Chapter IV begins with an explanation of the approach to fragmentation and reassembly processes in IPv6 and continues with the explanation of the fragmentation and reassembly part of the specification. The verification part of the specification is introduced in Chapter V. The thesis is summarized with a conclusions chapter. All PVS specification and proof files are provided in the appendices.

CHAPTER II

THE NEW INTERNET PROTOCOL (IPv6)

2.1 IPv6: The New Internet Protocol

The Internet is a set of networks including Arpanet, NFSnet, regional networks, local networks at a number of university and research institutions and number of military networks. The Internet Protocol (IP) is a network layer protocol that routes data across an internet [1]. To understand IP more precisely, it is convenient to start with an explanation of the Transmission Control Protocol/Internet Protocol (TCP/IP) because TCP/IP is a family of protocols [2] to which IP belongs.

TCP/IP is a group of protocols developed by a community of researchers centered around Advanced Research Projects Agency (ARPA) [2]. It allows computers of all sizes, using totally different operating systems, to communicate with each other via the Internet [3]. TCP/IP provides some well known services such as file transfer (FTP), remote login (TELNET) and e-mail, and it has a layered structure. In this layered structure, each layer has a different responsibility for network communication. Figure 2.1 shows the TCP/IP layers.

In this layered structure, the link layer, sometimes called the data link layer, includes the device driver in the operating system and the corresponding network interface card in the computer. The network layer handles the movement of the data packets around the network. IP, Internet Control Message Protocol (ICMP) and Internet Group Management Protocol (IGMP) are some of the members of this layer. The transport layer provides a data flow between two computers, for the application layer above. Transmission Control Protocol (TCP) and User Datagram

Protocol (UDP) are two important protocols in this layer. The application layer takes care of the details of the particular application. There are many common TCP/IP applications that every computer system provides. Some of the application in this layer are telnet, FTP, Simple Mail Transfer Protocol (SMTP), etc [5].

Application Layer	Telnet, FTP, e-mail, etc.
Transport Layer	TCP,UDP
Network Layer	IP, ICMP, IGMP
Link Layer	device driver and interface card

Figure 2.1: The layers of the TCP/IP protocols.

Figure 2.2 shows the flow of data between layers of the TCP/IP protocol suite. Data is prepared by a user and is sent to the application layer. Then with header information, it is sent to the transport layer. The transport layer takes this data, and it adds its header information to the data and then transfers it to the network layer. In the network layer, IP or some other protocol takes the data with the upper layer headers, adds its own header and sends it to the final layer. In the link layer, the link layer protocol takes all the data and header information and adds other information to the data, and finally, sends the resulting packet to the network. At this stage, whole headers and the data together are called a packet or a datagram. This whole process takes place in a computer that sends data through the network. After the packet is received in the receiving computer, it passes through all the layers again, but this time in the reverse order. In each layer, of the receiving computer,

the corresponding header information is cut from the packet. Finally, when the data reaches the application layer of the receiving computer, it is the same as the data sent by the transport layer of the sending computer [19]. The header information that is added to the data in each layer helps transmission of the data correctly in the network [12].

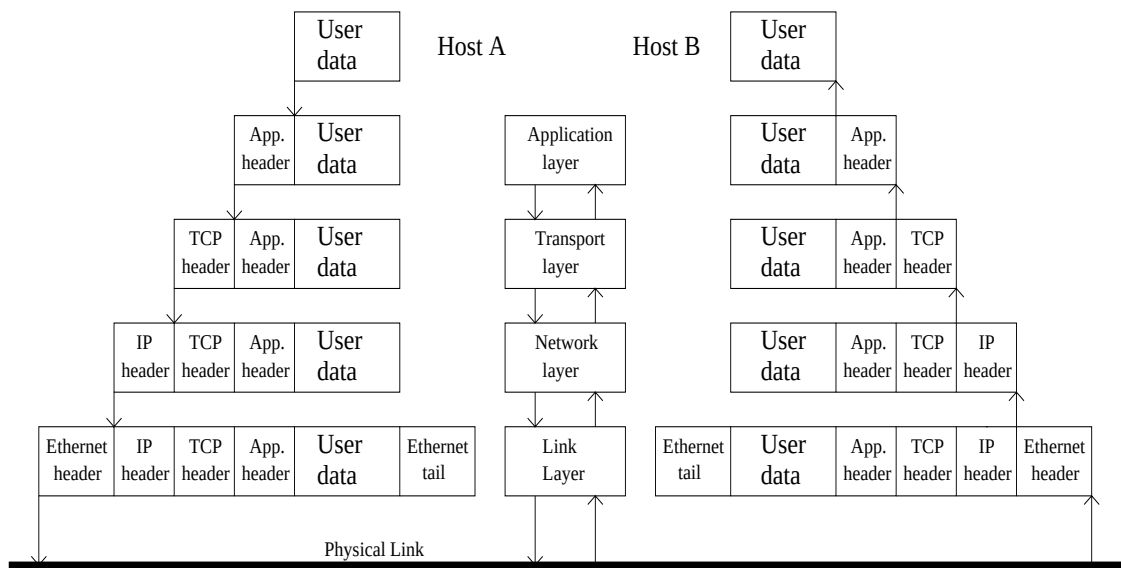


Figure 2.2: Flow of data through the layers of the TCP/IP protocols.

As was mentioned before, IP is one of the network layer protocols in the TCP/IP protocol suit. Its duty is to transmit all TCP, UDP and ICMP data across the network. IP provides an unreliable connectionless packet delivery between hosts. Packet delivery is unreliable because there are no guarantees that an IP packet successfully arrives at its destination. When something goes wrong, such as a router temporarily running out of buffers, IP has a simple error handling algorithm: throw away the packet and try to send an ICMP message to the source of the packet.

Beside being unreliable, IP is also a connectionless protocol. IP does not maintain any state information about successive packets. Each packet is handled independently from all other packets. This means that IP packets can be transmitted out of order. If a source sends two consecutive packets (first A, and then B) to the same destination, each is routed independently and can take different routes in the network. It is possible for packet B to arrive before packet A does [3].

Currently, in most TCP/IP implementations, the Internet Protocol version 4 (IPv4) is being used as the network layer protocol [22]. This protocol was first specified in 1980 [7]. The specification of the IPv4 was introduced in the Request For Comments 791 (RFC 791) [10]. As the Internet grew, it became apparent to many observers that the existing version of IP (IPv4) was insufficient to meet the performance and functional requirements for the Internet [19]. One of the reasons IPv4 became inadequate is that it is running out of network addresses [6]. Every computer in the Internet must have a network address and no two computers may share the same address. The current Internet Protocol has 32-bit address space to address hosts on the Internet. With 32-bit address space, it is possible to address a maximum of 4 billion computers on the Internet [9]. Due to the increasing popularity of the Internet, the number of computers on the Internet increases dramatically. Figure 2.3 depicts the number of hosts on the Internet by year. In 1993, the Internet Engineering Task Force (IETF) formed the Address Lifetime Expectations (ALE) Working Group to develop an estimate for the remaining life time of the IPv4 address space. ALE working group confirmed that, according to current statistics, the Internet would exhaust the IPv4 address space between 2005 and 2011 [14].

The other reason for IPv4 becoming inadequate is the explosion in the size of routing tables [6]. Routers are the devices that connect networks to create the

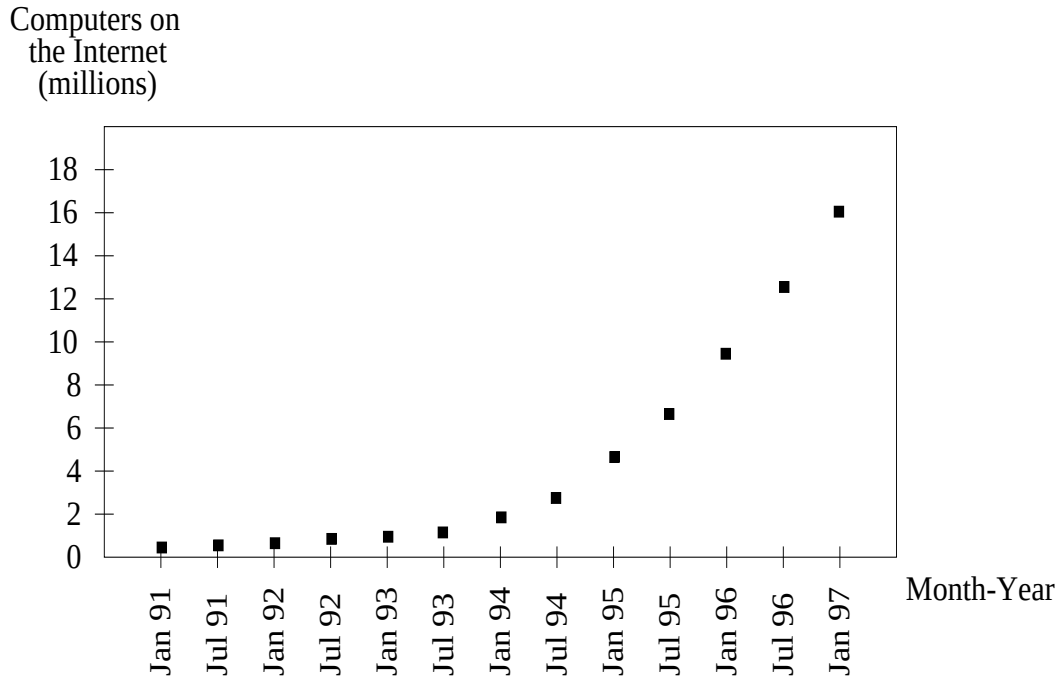


Figure 2.3: The number of hosts on the Internet by year [8].

Internet. These devices hold a routing table to track the location of computers connected to the network. As the number of connected computers grows, so does the size of these routing tables.

Due to these inabilities of IPv4, IETF began to search for options to replace IPv4 with a new version, one that would solve the problems of address exhaustion and routing table explosion. In 1993, the IETF formed the Next Generation IP (IPng) Area to evaluate the various proposals and select a successor to IPv4 [7, 6]. (When the new Internet Protocol was first specified, it was called the Internet Protocol Next Generation (IPng)). The new Internet Protocol is now officially called the Internet Protocol version 6 (IPv6). Internet Protocol version 6 was recommended by the IPv6 Area Directors of the IETF at the Toronto IETF meeting on July 25,

1994 and documented in RFC 1752, “The recommendation for the IP next generation Protocol”. The recommendation was approved by the Internet Engineering Steering Group in November 1994 [22]. The final specification of IPv6 was introduced in RFC 1883 [15].

Changing the Internet Protocol takes more than simply changing a single protocol. Changes to IP affect many other TCP/IP protocols. In fact, at least 58 current TCP/IP standards have to be revised to accommodate IPv6 [6, 9]. Today, there are literally millions of systems using IPv4. Most likely, some of those systems will never convert to IPv6, and it will take several years to upgrade those systems that do change [6].

IPv6 has been designed as an evolutionary step from IPv4. The functions which are generally seen as working in IPv4 were kept in IPv6. Functions which do not work or were infrequently used were removed or made optional. In the following sections, some of the new features of IPv6 such as the header format and fragmentation and reassembly will be introduced. The scope of this thesis study is the formal specification and verification of packet fragmentation and reassembly in IPv6; therefore, fragmentation and reassembly in IPv6 will be introduced more precisely.

2.1.1 IPv6 Header Format

The Internet Protocol treats each message independently, forwarding it through the network to its final destination. By convention, the messages that IP transfers are called datagrams or packets. Each IP packet begins with a common header format [6]. Figure 2.4 shows a full IPv6 packet with all extension headers [19]. In this section, the header format of IPv6 will be introduced.

Length in octects:	
IPv6 Header	40
Hop_by_hop options header	Variable
Routing header	Variable
Fragment header	8
Authentication header	Variable
Encapsulation security payload header	Variable
Destination options header	Variable
TCP header	20 (optional variable part)
Application data (User data)	Variable

Figure 2.4: The IPv6 packet with all the extension headers.

The header section of an IPv6 packet consists of two sections, the basic IPv6 header and the IPv6 extension headers which are hop-by-hop header, routing header, fragment header, authentication header, encapsulation security payload header and destination options headers. Figure 2.5 depicts both the IPv6 header and the IPv4 header [15, 2].

The IPv6 header must appear at the beginning of each IPv6 packet [9, 19]. Fields of the basic IPv6 header are explained as follows:

- **Version (4 bits):** This field holds the version number of the IP header. For IPv6, the version number is 6 [19, 15].
- **Priority (4 bits):** This field enables a source to identify the desired transmit and delivery priority of each packet relative to other packets from the same source [19].

Version	Prio.	Flow Label		
Payload Length		Next Header	Hop Limit	
Source Address				
Destination Address				

IPv6 Header

Version	IHL	Type of Service	Total Length	
Identification			Flags	Fragment Offset
Time to Live		Protocol	Header Checksum	
Source Address				
Destination Address				
Options				Padding

IPv4 Header

Figure 2.5: The IPv6 and IPv4 header formats.

- **Flow Label (24 bits):** This field may be used by a source host to label those packets for which it requests special handling by the IPv6 routers, such as non-default quality of service or “real time” service [15].
- **Payload Length (16 bits):** Length of the remainder of the IPv6 packet following the IPv6 header, in bytes is held in this field. Zero value in this field indicates that the payload length is carried in a jumbo payload hop-by-hop option [15, 7].
- **Next Header (8 bits):** It identifies which header follows the IPv6 header in the packet. It may indicate an optional IP header or an upper layer protocol.

There are some specific values set for this field and each value identifies a different next header. These values are also used in the next header field of the IPv6 extension headers [6].

- **Hop Limit (8 bits):** This field determines how far a packet will travel on the Internet. When a host creates a packet, it sets the hop limit to some initial value. Then as the packet travels through routers on the Internet, each router decrements value of this field by one. If the hop limit of the packet becomes zero before it reaches its destination, the packet is discarded by a router. The hop limit field serves two purposes. The first purpose is to break routing loops and the second, to let a host perform an expanding search across the network [6].
- **Source Address (128 bits):** The address of the source of the packet is held in this field [15].
- **Destination Address (128 bits):** The address of the intended recipient of the packet is held in this field [15].

When we compare the two headers, IPv4 and IPv6, we see that the IPv6 header is much simpler than the IPv4 header. There are only six fields and two address fields in IPv6, where the IPv4 header has 10 fixed header fields, two address fields and some options [9]. Although the IPv6 addresses are four times longer than the IPv4 addresses, the IPv6 header is only twice the size of the IPv4 header [22].

The IPv6 headers do not contain any optional elements. This does not mean that it is impossible to express options for special-case packets. Instead of the option fields in IPv4, extension headers are added after the main IPv6 header. In IPv6, it is possible to add an arbitrary number of extension headers between the IPv6 header and the payload of the packet. Each header is identified by a header type and carries

the header type of the following header in the chain or that of the payload in the case of the last extension [9]. Figure 2.6 shows all extension header formats. Extension headers of IPv6 are:

Next Header	Hdr. Ext. Len.	
Options		

Hop_by_hop Header

Next Header	Hdr. Ext. Len.	
Options		

Destination Options Header

Next Header	Hdr. Ext. Len.	Reserved
Security Parameter Index		
Authentication Data		

Authentication Header

Next Header	Hdr. Ext. Len.	Routing Type	Segment Left

Generic Routing Header

Next Header	Hdr. Ext. Len.	Routing Typ=0	Segments Left
Reserved	Strict/Loose Bit Map		
Address[1]			

Address[n]			
------------	--	--	--

Routing Header

Vers.	Pri.	Flow Label	
Payload Length		Next Header	Hop Limit
Source Address			
Destination Address			
Security Parameter Index			
Encrypted Payload			

Encapsulating Security Payload Format

Next Header	Reserved	Fragment Offset	Resm
Identification			

Fragment Header

Figure 2.6: The IPv6 extension header formats.

- **Hop-by-hop Header:** This extension header is used to carry optional information. The information carried by this header must be examined by every node along a packet's deliver path [15].
- **Routing Header:** The routing header contains a list of one or more intermediate nodes to be visited on the way to a packet's destination. It plays the same role as the source option of IPv4 [19]. Figure 2.6 shows the generic routing header format. In addition to this general routing definition, Type 0 routing header is also defined in RFC 1883 [15]. Type 0 routing header is also shown in Figure 2.6.
- **Fragment Header:** Fragmentation in IPv6, may only be performed by a source host. When a packet's length is larger than the maximum transmission unit (MTU) of that packets' route, the source host divides the packet into fragments and adds a fragment header to each fragment [19].
- **Destination Options Header:** To carry optional information examined only by the packets destination host, the destination options header is used by the IPv6. This option header may appear in a packet more than once [19].
- **Authentication Header:** This header provides support for data integrity and authentication of the IPv6 packets [19].
- **Encapsulating Security Payload Header:** The use of the encapsulated security payload header supports the privacy and data integrity for the IPv6 packets.

An IPv6 packet may contain more than one extension headers. The IPv6 specification (RFC 1883) suggests the following order for the extension headers [15]:

1. The basic IPv6 Header
2. Hop-by-hop Header

3. Destination Options Header (1)
4. Routing Header
5. Fragment Header
6. Authentication Header
7. Encapsulating Security Payload Header
8. Destination Options Header (2)

2.1.2 Fragmentation and Reassembly

The networks in the Internet are very dissimilar since each of them is designed to serve users with a variety of different needs, by employing many diverse communications media. One of the most important differences between networks is the maximum packet size allowed by each network.

This packet size is called the Maximum Transmission Unit (MTU). Table 2.1 shows various networks and their MTU sizes [26, 13]. Later in this section, MTU and the path MTU will be explained. Different MTU values for each network creates a problem. The problem is what to do when a large packet must cross a network with a smaller MTU. This problem in network interconnection can be solved in two ways. The first way is to avoid the problem, e.g. by sending only small internetwork packets. The second way is to split the packet into smaller pieces. This technique is called packet fragmentation [21].

There are two fragmentation strategies, the intra-network fragmentation and the inter-network fragmentation. In the intra-network fragmentation, oversize packets are fragmented in a network by the entrance gateway, a device that routes packets, and later reassembled by the exit gateway. Fragmentation is done locally, and is therefore both temporary and transparent to other networks. The current IP standard does not support the intra-network fragmentation strategy [20]. On the

other hand, inter-network fragmentation is global and permanent, and is supported by the IP standard. Once a packet is fragmented, each fragment is seen as a new packet, and the original packet is reassembled by the destination host only [23]. One important property of the inter-network fragmentation is that of dynamic routing of fragments. This means that the fragments of a packet can follow similar or dissimilar routes from the point of fragmentation to the final destination [20].

<i>Network Type</i>	<i>MTU</i>
Hyperchannel	65535 bytes/packet
16 Mbits/sec token ring (IBM)	17914 bytes/packet
SMDS	9180 bytes/packet
IEEE 802.4	8166 bytes/packet
FDDI	4500 bytes/packet
4 Mbits/sec token ring (IEEE 802.5)	4352 bytes/packet
ProNET-10	2044 bytes/packet
Wideband Satellite Net	2000 bytes/packet
Ethernet	1500 bytes/packet
IEEE 802.3/802.2	1492 bytes/packet
X.25	576 bytes/packet
Point-to-point	296 bytes/packet
Packet Radio Net	254 bytes/packet
ARPANET	126 bytes/packet
ALOHANET	80 bytes/packet

Table 2.1: Various networks and their respective MTU sizes.

Both fragmentation strategies have advantages and disadvantages. For example, the intra-network fragmentation is transparent and network-specific, and thus can minimize the overhead incurred by the fragmentation in that network. On the other hand, a packet's fragments must all reach the same gateway or router.

No strategy is always better than the other, and its choice is an important design decision. Most of the commercial and some public networks like the Xerox Network System (XNS) and the IBM System Network Architecture (SNA), adopt an intra-network fragmentation, while the inter-network fragmentation is used in The Federal Research Internet (FRI) system, which includes ARPANET and other networks [24, 25].

2.1.2.1 MTU and Path MTU

As it is seen from table 2.1, there is a limit on the size of a packet for each network type. This limit is called the Maximum Transmission Unit (MTU). If IP has a packet to send, and the size of a packet is larger than the MTU, IP performs fragmentation, breaking the packet up into smaller pieces, so that each fragment is smaller than the MTU or equal to MTU [4]. Fragmentation and reassembly of packets will be discussed later.

When two hosts on the same network are communicating with each other, they only need to know the MTU of the networks that they belong. When two hosts are communicating with each other across multiple networks, each link can have a different MTU. The important values are not the MTU values of the two networks to which the two hosts are connected, but rather the smallest MTU of any data link that packets traverse between the hosts. This is called the path MTU [13].

The path MTU between any two hosts depends on the route used at any time; therefore, it does not need to be constant. Also, routing does not need to be symmetric. This means that the route from A to B may not be the reverse of the route from B to A [3].

In the IPv4 network architecture, both routers and hosts are capable of fragmenting packets. This approach reduces the burden on the source of a packet, as the

sender does not have to worry about the MTU size along the path. If the packet is too big to cross a link, the routers take care of the required fragmentation. Although IPv4 seems that it does not need to know the path MTU, sometimes a host wants to send packets as big as possible. In such a case, IPv4 needs to know the path MTU. A protocol, the Path MTU discovery protocol, was developed for IPv4 to discover the path MTU. This protocol was documented in RFC 1191 [13].

On the other hand, IPv6 strongly needs to know the path MTU because IPv6 does not provide hop-by-hop fragmentation. This means that fragmentation takes place only at the source node not at the intermediate nodes. A new path MTU discovery technique was developed for IPv6. This technique is largely derived from the IPv4 path MTU discovery protocol. The basic idea is that initially a source node supposes that the path MTU of a path is the MTU of the first hop in the path. If any of the packets sent on that path are too big to be forwarded by some node along the path, that node will discard the packets and return an ICMPv6 (Internet Control Message Protocol version 6) “Packet Too Big” message [17]. After receiving such a message, the source node reduces its supposed path MTU for the path based on the MTU of the constricting hop as reported in the “Packet Too Big” message. Several iterations of the “Packet-sent/Packet-too-big-message-received” cycle can occur before the path MTU is discovered, as there may be links with smaller MTUs further along the path.

Because of the changes in the routing topology, the path MTU of a path may change over time. Reductions of the path MTU are discovered by a “Packet Too Big” message. In order to detect increases in a path’s path MTU, a node periodically increases its assumed path MTU. This will almost always results in packets being discarded and “Packet Too Big” message being generated, because in most cases the

path MTU of the path will not have changed; therefore, hosts should not attempt to detect a path's MTU frequently [16].

2.1.2.2 Fragmentation and Reassembly in IPv4

Whenever the IPv4 layer receives an IPv4 packet to send, first it determines which local interface the packet is being sent on, then it requires that interface to obtain its MTU. After that, IPv4 compares the MTU value with the size of packet and performs fragmentation, if necessary. As was mentioned before, in IPv4 architecture, fragmentation can be performed either at the original sending host or at an intermediate router.

When an IPv4 packet is fragmented, the fragments are not reassembled until they reach their final destination. The IP layer at the destination of fragments performs the reassembly process to reassembly fragments in order to build original packet. The goal is to make fragmentation and reassembly processes transparent to the upper layer (TCP and UDP), except for possible performance degradation. In an IPv4 network, it is also possible for a fragment of a packet to be fragmented again (possibly more than once). The information maintained in the IPv4 header for fragmentation and reassemble provides enough information to do this [3].

The literature includes algorithms for fragmentation and reassembly of the IPv4 packets. Two algorithms were introduced in RFC 791, the Internet Protocol and Darpa Internet Program Protocol Specification [10]. Another reassemble algorithm was introduced in RCF 815, the IP packet reassembly algorithms [11].

2.1.2.3 Fragmentation and Reassembly in IPv6

One of the most important differences between IPv4 and IPv6 fragmentation is the place of the fragmentation. Unlike IPv4, fragmentation in IPv6 is performed only by the source nodes, not by routers along packet's delivery path. As was explained

before, in IPv4 a packet can be fragmented and reassembled by a router along the packets path. If necessary, even a fragment of an packet can be fragmented and reassembled. On the other hand, in IPv6, a packet may be fragmented just once by source nodes and the fragments can be reassembled by destination hosts [14].

The other difference between IPv4 and IPv6 fragmentation is the fragmentation header. IPv4 does not have a fragmentation header, in fact, IPv4 has just one header. All necessary information for fragmentation and reassembly, identification, flags and fragment offset, is included in this header. In IPv6, this information is kept in an extension header called the fragment header. The IPv6 fragmentation extension header was introduced in section 2.1.1. Figure 2.6 illustrates the fragment header. The fields of the header are:

- **Next header (8 bits):** Identifies the type of header immediately following this header. The header that follows the fragment header is considered the first header in the fragmentable part of a packet. The meaning of fragmentable part will be explained later [15].
- **Reserved (8 bits):** This field is reserved for future use. It is set to zero for transmission and ignored on reception.
- **Fragment offset (13 bits):** This field indicates where in the original packet the payload of this fragment belongs. The payload is measured in 8-byte units. This implies that fragments (other than the last fragment) must contain a data field that is a multiple of 8 bytes long [9].
- **Res (2 bits):** This field is reserved for future use and it is set to zero for transmission and ignored on reception.
- **M flag (1 bit):** Zero in this field indicates that this is the last fragment and one indicates that there are more fragments.

- **Identification (32 bits):** This field holds an identification value to uniquely identify the original packet. The identification must be different than that of any other fragmented packet recently sent with the same source address and destination address. To meet this requirement, a 32-bit “wrap-around” counter is used, and it is incremented by one each time a packet must be fragmented [15].

To take full advantage of the internetworking environment, a node with an IPv6 implementation must perform a path discovery algorithm that enables it to learn the smallest MTU supported by any subnetwork on a path. With this information, the source node fragments packets, as required, for each given destination address. Otherwise, the source must limit all packets to 576 bytes, which is the minimum MTU that must be supported by each subnetwork for IPv6 [19].

The initial unfragmented packet is called the original packet. IPv6 recognizes that the original packet has two parts, the fragmentable part and the unfragmentable part. Figure 2.7 displays the original packet.

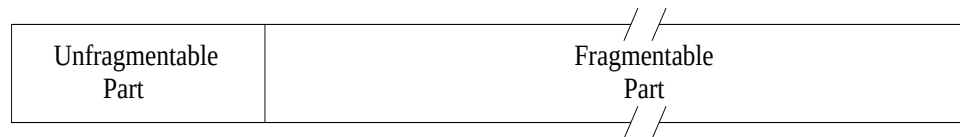


Figure 2.7: The unfragmented original packet.

The first step in the fragmentation process is to identify the fragmentable and unfragmentable parts of the original packets. The unfragmentable part includes the IPv6 header plus any extension headers that must be processed by nodes along the route of the fragments. The following extension headers, if present, must be in the

unfragmentable part: hop-by-hop header, routing header, and a destination options header. The rest of the extension headers that need to be processed only by the final destination node and payload of the original packet are considered the fragmentable part [18]. The relationship between the size of the unfragmentable part and the fragmentable part is given in the equation (2.1).

$$size_of_unfragmentable = size_of_packet - size_of_fragmentable \quad (2.1)$$

The fragmentable part of the original packet is divided into fragments. Each fragment's length, except for the last fragment, is arranged so that the length of each fragment becomes a multiple integer of 8 bytes long, and the resulting fragment packets' size must fit within the MTU of the original packet's path. Figure 2.8 shows the original packet after the fragmentable part is divided into fragments [15].

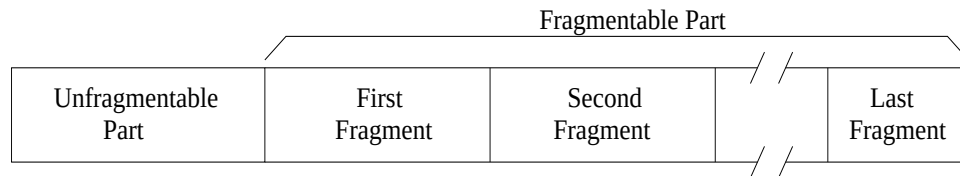


Figure 2.8: The original packet after dividing into fragments.

After dividing the fragmentable part into fragments, the fragment header is appended to the unfragmentable part, and then each fragment is appended to a copy of the unfragmentable part plus the fragment header. Figure 2.9 depicts the fragmented packets [15].

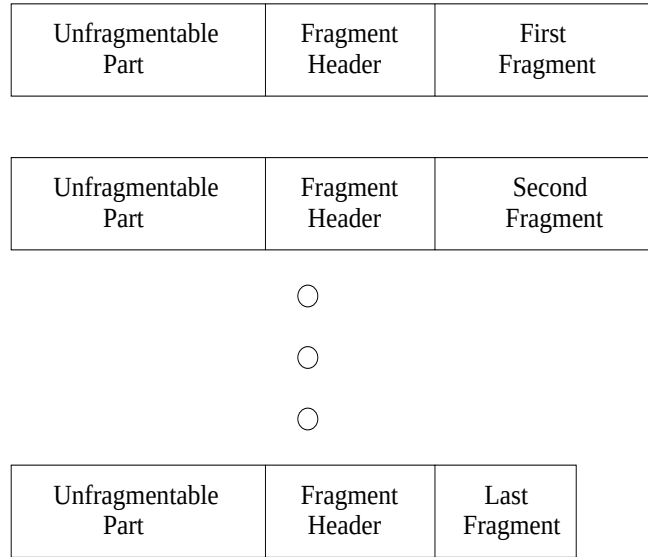


Figure 2.9: The fragment packets.

Each fragment packet is composed of the unfragmentable part of the original packet, a fragment header and the fragment itself. When the fragment header is appended to the unfragmentable part, two fields in the unfragmentable part must be updated. First, the payload length field of the IPv6 header must be updated to reflect the length of the fragment packet. Second, the next header field in the last header of the unfragmentable part must be changed to indicate that a fragment header follows. The next header value in the last header of the unfragmentable part must be kept in the next header field of the fragment header so that the destination node can understand the first extension header, if there is any, in the fragmentable part [18]. The M flag in the fragment header must be set to 1 for each fragment packet except for the last one. In the last fragment packet, this flag must be set to zero to indicate that this is the last fragment packet. Figure 2.10 illustrates an example of the IPv6 fragmentation.

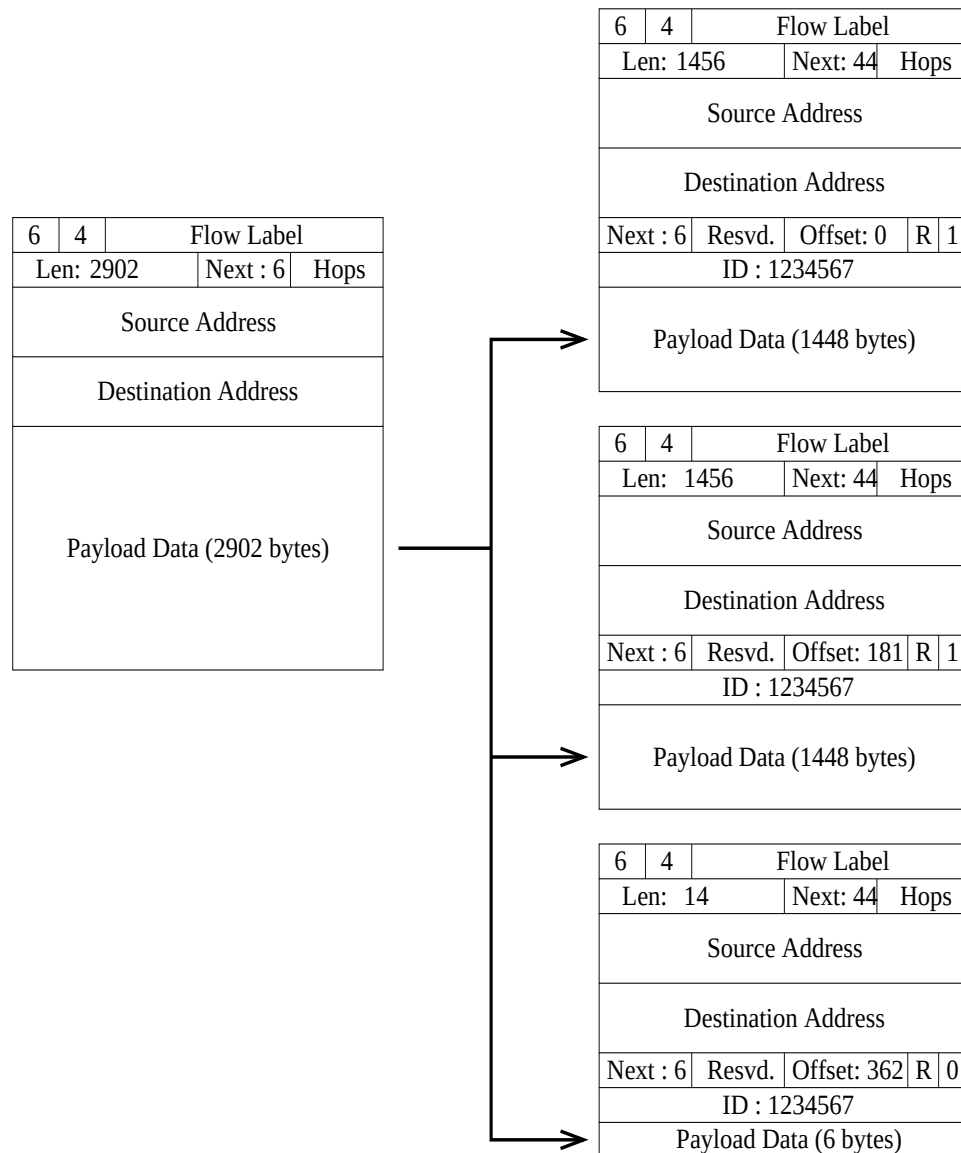


Figure 2.10: The IPv6 fragmentation example.

For simplicity, just the basic IPv6 header is included in the figure; therefore, the payload section of the original packet can be considered as the fragmentable part and the basic IPv6 header can be considered as the unfragmentable part.

The original packet which is to be transmitted across an Ethernet Local Area Network (LAN) has 2902 bytes of payload data. Since Ethernet frames can only carry 1500 bytes of data (including the basic IPv6 header), fragmentation is necessary. Even though 2902 bytes is less than twice the 1500-byte limit, a third fragment is needed. There are two reasons for the third fragment. The first reason is the overhead of the basic IPv6 header and extension headers, if there is any. The second and real reason is IPv6's restriction on fragment sizes. Since the first fragment must contain multiples of 8 bytes, it cannot completely fill a 1500 byte ethernet frame. Instead, IPv6 must settle for 1496 bytes. When 48 bytes of basic IPv6 header and fragment header overhead is subtracted from 1496, it is equal to 1448. This means that, for payload data, there are only 1448 bytes available. The same restriction is applied to the second fragment, so again 1448 payload bytes are included in the second fragment. The final fragment, the third fragment, includes the remaining 6 bytes of the payload [6].

The reassembly process takes place at the destination host of the fragment packets. When the destination host reassembles them, it identifies the fragments that belong to the same packet by looking at the source, destination IPv6 addresses and the identification value inserted in the fragment header. Individual fragments are queued within the network layer until the original packet can be completely reassembled. At this point, it is passed to the appropriate protocol module.

When all fragments have arrived, the original packet can be reassembled. A single copy of the unfragmentable part of the first fragment is kept to construct the unfragmentable part of the reassembled packet. The fragmentable part of the reassembled packet is constructed from the fragments following the fragment header in each of the fragment packet. While constructing the fragmentable part of the

reassembled packet, all fragment packets are ordered according to their offset values in their fragment headers. The ordering process is done in order to place each fragment to its correct place in the fragmentable part [15].

The payload length field of the basic IPv6 header is updated to reflect the length of the reassembled packet. The payload length of the reassembled packet is computed from the length of the unfragmentable part and the length and offset of the last fragment. The next header field of the last extension header in the fragmentable part is also updated to point the first extension header in the fragmentable part. It is updated from the next header field of the fragment header of the first fragment packet [18].

During the reassembly process, when the first fragment of a packet arrives at the destination, the destination host starts a timer and it sets the timer to 60 seconds. If the timer expires before all the fragment packets arrive, the fragments are discarded and the reassembly process is abandoned. If the first fragment has been received in the first 60 second, an ICMP “Time Exceed/Fragment Reassembly Time Exceed” message is sent to the source of that fragment packet [15].

In two other cases, IPv6 discards fragment packets and sends an ICMP “Parameter Problem Code 0” message to the source of the fragments. The first case is that if the length of a fragment is not a multiple of 8 bytes and the M flag of that fragment is set to 1. The second case is that if the length and offset of a fragment are such that the Payload Length of the packet reassembled from that fragment would exceed 65,535 bytes [15].

CHAPTER III

FORMAL METHODS IN COMMUNICATION PROTOCOLS

In this thesis study, IPv6 fragmentation and reassembly processes are formally specified and verified in the Prototype Verification System (PVS) environment. For that reason, in this chapter, first formal methods are introduced and then PVS and its features are explained. Then, some examples of formal specification and verification studies are introduced. Finally, a specification of IPv6 headers is presented as the basis of this thesis.

3.1 Formal Methods and PVS

3.1.1 Formal Methods

The term formal methods refers to the application of mathematical techniques for the specification, analysis, design, implementation and subsequent maintenance of complex computer software and hardware [27]. The most important advantage of formal techniques is that a formal specification is a mathematical object which has an unambiguous meaning. For that reason, mathematical methods may be used to analyze these specifications, such as formal verification of the correctness and completeness of a specification.

To derive test cases automatically, a formal specification can be used to check whether a particular implementation behaves as expected. If the implementation language is a formal language as well, it even makes it possible to verify formally that an implementation satisfies a given specification [28].

Formal methods have been applied in the design of a great number of systems. Using formal notation increases the understanding of the operation of a system, especially early in a design. By using formal methods, design choices can be explored. Such methods assist the design team in thinking about the operation of the system before it is implemented. Missing parts of an incomplete specification became more obvious. The remaining parts of a design can be identified, and alternative possibilities can be considered. In particular, error conditions can be checked by calculating the precondition of an operation.

One of the most important considerations in an industry is lowering the overall cost of an product. Errors corrected at the design stage can be up to two orders of magnitude cheaper to correct, than if they are found later. The Pentium FDIV bug can be given as an example to this situation which attracted a great deal of public interest [29]. It caused Intel to take a \$475 million charge against revenues. This kind of error can be prevented by using formal methods. The initial barrier of using formal methods is the notation, which may contain unfamiliar symbols, and will require designers to attend training courses [30]. However, in general, the notation is no worse than learning a new style of programming language.

The first step in using formal methods is specifying a system in a specification language. The specification can be functional or behavioral depending on the system specified. After a system is specified and an implementation is built, the second step is the verification of the system. The goal of verification is to ensure that the right implementation has been built, i.e., that the behavior of the implementation is what was intended. Verification not only applies to an implementation, which must be checked against the specification, but also to a specification itself. The specification must be complete and consistent, and satisfy critical application properties [31].

For most formal methods, software tools have been developed that assist in analyzing a given specification. Apart from syntax-checking and type-checking, most software environments for formal methods also provide a means for rapid prototyping. Even automatic generation of an implementation is sometimes possible. Some other tools can generate test sequences that can be used for conformance testing, which is an important validation activity in constructing software [28].

Today a number of different formal specification techniques exist, some of which are general purpose such as PVS [33], Z [30] and Vienna Development Method (VDM) [41], while others are generally used in a specific domain of application such as LOTOS [42] and Specification and Description Language (SDL) [43]. These languages are based on some mathematical theories such as set theory, temporal logic and lambda-calculus, and process algebra [28].

3.1.2 PVS: Prototype Verification System

As was mentioned in the previous section, there are a variety of specification environments available today. PVS is one of these specification environments. In this thesis study, the PVS environment was used to specify the fragmentation and reassembly processes in IPv6. Thus, in this section, PVS is introduced.

PVS is an integrated environment for constructing and analyzing clear and precise formal specifications and for developing readable proofs. It supports a wide range of activities some of which are creating, analyzing, managing and documenting theories and proofs. PVS was designed by researchers at SRI International Computer Science Laboratory [32, 33]. The PVS environment consists of a specification language, a parser, a typechecker, a prover, specification libraries, and various browsing tools.

A PVS specification consists of one or more specification files, each of which contains one or more theories or datatypes. For each specification file, PVS creates a number of extra files to keep track of the status of the specification. If the name of the specification is `a_spec` then PVS keeps the specification itself in `a_spec.pvs`. To keep track of the proof of specification and to save the proof commands that are used to prove the specification, PVS creates another file called `a_spec.prf`. Every time the user tries to prove a theory in the specification, this file is automatically updated by PVS. Another file created by PVS is `a_spec.bin`. In this file, a binary form of the typed specification is kept. Another feature of the PVS is creating dump files. Into a dump file of a specification, PVS puts all necessary information related to the specification including the proof file, the specification itself, the `.bin` file and other specifications imported by this specification. The purpose of creating dump files is to make it easy to move PVS specifications. In a directory, there can be a number of specifications to build a whole specification of a system. To keep track of the specification files in a directory, PVS creates `.pvscontext`. Just one context file is created for each directory and is updated automatically by PVS every time a user works in the directory [33].

Users of PVS can check theories for syntactic consistency by using the parser function of PVS. When a theory is parsed, PVS builds an internal representation that is used by the other components of the system. The typechecking function of PVS analyzes theories for semantic consistency and adds semantic information to the internal representation built by parser. Theorem proving may be required to establish the type-consistency of a PVS specification because the type system of PVS is not algorithmically decidable. The theorems that need to be proved for type consistency are called *type-correctness conditions* (TCCs). TCCs are also attached

to the internal representation of the theory and displayed on request. Some TCCs are proven during typechecking. For TCCs that are not proven during typechecking, PVS requires assistance from the user [34].

PVS has been used to verify a variety of examples from functional programming, fault tolerance, and real time computing. One of the most important applications of PVS is in the verification of the microcode for selected instructions of a commercial-scale microprocessor called AAMP5, designed by Rockwell-Collins and containing about 500,000 transistors. Most recently, PVS has been applied to the verification of the design of an SRT divider [44] (SRT divider was discovered by D. Sweeney of IBM and this name was given to it by C. V. Freiman [35]. This divider was used in the design of Intel's Pentium processor [29].) [32]. Recently, in a research study, Paul and Leathrum extended the PVS treatment of SRT divider to include IEEE floating point standards [45].

3.1.2.1 The PVS Specification Language

The PVS specification language is based on classical, simply typed, higher-order logic [32]. Starting from the base types (booleans, rationals, integers, etc.) types can be defined in a theory by using the function, record and tuple type constructions.

PVS specifications are just text files including a sequence of lexical elements. The lexical elements of PVS can be listed as identifiers, reserved words, special symbols, numbers, whitespace characters and comments. Comments must begin with “%” character and end with a newline.

A PVS specification consists of a collection of theories. Each theory must have a signature for the type names and constants declared in the theory. Axioms, definitions, and theorems declared in the theory are associated with the signature.

A theory can use other theories and other definitions declared in other theories, by importing them. It is also possible to pass values and parameters defined in certain type to a theory.

In PVS, declarations are used to introduce types, variables, constants, and formulas. A PVS declaration consists of an identifier, an optional list of bindings, and a body. The body of a declaration determines the kind of the declaration. The bindings and the body together determine the signature and the definition of the declared entity. PVS allows the overloading of declaration identifiers. Thus, in a theory, a constant, a formula and a function can take the same name [36].

PVS allows four different kind of type declarations. These are uninterpreted type declaration, uninterpreted subtype declaration, interpreted type declaration and enumeration type declaration. Besides type declarations, variables and constants of any type can be declared. Formula declarations in PVS introduce axioms, lemmas, theorems, assumptions, and obligations. During the proof session, the identifier associated with the formula declaration may be used. Axioms, lemmas, theorems, assumptions, and obligations are introduced with the keywords **AXIOM**, **LEMMA**, **THEOREM**, **ASSUMPTION** and **OBLIGATION**, respectively.

The PVS specification language offers the user usual expression constructs, including logical and arithmetic operators, quantifiers, lambda abstractions, function applications, a polymorphic **IF-THEN-ELSE** and function and record overrides. The language has a number of predefined operators. All of the infix operator can also be used in prefix form. For instance, $(x + 1)$ and $+(x, 1)$ are equivalent [36].

3.1.2.2 The PVS Proof Checker

After successfully parsing and typechecking a PVS specification, the final step is proving the specification. The user of the specification should provide lemmas

and theorems about the system specified. To prove TCCs, axioms, and the user's lemmas and theorems related to the specification, PVS provides a built-in proof checker.

The main function of the PVS proof checker is to construct readable proofs. PVS pays great attention to simplifying the process of developing, debugging, maintaining and presenting proofs. To make proof developing easier, the PVS proof checker provides a number of powerful proof commands to carry out propositional, equality and arithmetic reasoning with the use of definitions and lemmas. To form proof strategies, these proof commands can be combined. The PVS proof checker allows proof steps to be undone in order to make proof easier, and it also allows the specification to be modified during the course of a proof. PVS also allows a proof to be edited and rerun to support proof maintenance [37].

The PVS prover interacts with the user during a proof session, although the prover supports a batch mode in which proofs can be rerun. The prover maintains a proof tree, and the goal of the user is to construct a proof tree which is complete, in the sense that all of the leaves of the tree are recognized as true. Each node of the proof tree is called a proof goal. Each proof goal is called a sequent and consists of a sequence of formulas called antecedents, and a sequence of formulas called consequents. PVS displays such a sequent as follows:

$$\begin{array}{ll}
 \{-1\} & A_1 \\
 \{-2\} & A_2 \\
 [-3] & A_3 \\
 & \vdots \\
 |-----| & \\
 \{1\} & B_1 \\
 [2] & B_2 \\
 \{3\} & B_3 \\
 & \vdots
 \end{array}$$

where A_i are the antecedents PVS formulas and B_j are the consequent PVS formulas. To separate the antecedents from the consequents, a row of dashes is used. The intuitive meaning of a sequent is that the conjunction of the antecedents should imply the disjunction of the consequents, i.e.

$$(A_1 \wedge A_2 \wedge A_3 \cdots) \supset (B_1 \vee B_2 \vee B_3 \cdots) \quad (3.1)$$

The proof tree starts off with a root node of the form $\vdash A$, where A is the theorem to be proved. By adding subtrees to leaf nodes, PVS proof steps build a proof tree. It is easy to realize that a sequent is *true* if any antecedent is same as any consequent, if any antecedent is *false*, or if any consequent is *true*. Once a sequence is recognized as *true*, the branch of the proof tree is terminated. The goal is to build a proof tree whose branches are all terminated in this way. In the sequent displayed above, numbers in braces such as $\{3\}$, as opposed to brackets, highlights those formulas that are changed from those of the parent sequent [37].

PVS commands can be used to present lemmas, expand definitions, apply decision procedures, eliminate quantifiers, and so on; they affect the proof tree. The proof commands are saved when the proof is saved. There are two ways to invoke proof command. In one way, the user invokes the commands directly. The other way to enter a command is executing a proof strategy. The action resulting from a proof command is called *proof step* or *proof rule*. The proof commands that really introduce the PVS logic are called *primitive rules*. They can either identify the current sequent as *true* and terminate that branch of the proof tree, or they add one or more child nodes to the current sequent and transfer the focus of the user

to one of those child branches. Besides single proof commands, PVS has *strategies* which are combinations of proof steps that can add a subtree of any depth to the current node. On the other hand, those proof steps silently reduce those branches of the subtrees which they generate are identified as *true*, and collapse all remaining interior nodes, so that the subtree actually generated has depth zero or one [37].

One of the important features of the PVS prover commands is that its proof steps are extremely sophisticated. These proof steps can employ arithmetic and equality decision procedures. Several PVS proof commands are available.

3.1.3 Formal Specification and Verification Studies

It is possible to find many formal method studies related to communication protocol specification and verification. In this section, some of these studies are introduced as examples. Three studies by Tat Y. Choi, James K. Huggins, and Milica Barjaktarovic will be introduced. Finally, a study by Leathrum, a study for which this thesis is based, will be discussed.

In a paper, Tat Y. Choi discussed formal techniques for the specification, verification, and construction of communications protocols. He put formal methods for specifying protocols into three main categories. The first category includes transition models such as finite state machine and petri net models. The second category includes language-oriented models such as formal language and programming language models. In the third category are hybrid models which include both states and language constructions in the specification of protocols. In his paper, he concentrated on the finite state machine model and an extension of finite state machine model for protocol specification and verification. He said that protocols can be modelled by event driven processes which communicate with each other through message passing. The communication channels between processes can be modeled as first in

first out (FIFO) queues, and the protocol processes can be modeled as finite state machines. To model protocol processes, an abstract machine model which is a generalization of the finite state machine model, can also be used. For verification of the protocols, he discussed the reachability analysis method and the deductive inference method for the finite state machine model and the abstract machine model respectively. Finally, in the conclusion, he stated that the methods that were discussed in the paper are general enough that they can be applied to protocols at any layer of the Open System Interconnection (OSI) Reference Model [38].

Kermit is a popular communication protocol. In a study, James K. Huggins formally specified Kermit and verified it. His main goal was a faithful readable specification which allows anyone to formalize the intuitive verification proof without much overhead. He used evolving algebra approach. He began with evolving algebra specifications and verifications of two more abstract communications protocols used by various versions of Kermit: the alternating bit protocol and the sliding window protocol. He said that a nice feature of the evolving algebra approach is that the road from an intuitive proof to a precise one is very short; there is little overhead. He presented a series of evolving algebras for the Kermit protocol, filling in the pieces where it was necessary to show how Kermit uses the abstract protocols. As usual with the protocols, he proved theorems dealing with properties of safety and liveness. His safety theorems are of the form “Every state reachable in any relevant run satisfies property Φ ” and are proved by induction on relevant runs. The liveness theorems have the form “Every fair run has such and such property” [39].

In his research, Milica Barjaktarovic specified and verified the Open System Interconnection (OSI) Session Layer (SL). As a formal tool, he used Milner’s process algebra called the Calculus of Communicating Systems (CCS). He modeled both the

Session Layer service and Session Layer protocol in CCS, and verified it using CCS's automated model checker. He verified that the protocol specification satisfies the service specification [46].

In a study, Dr. Leathrum et al. specified and verified parts of the new Internet Protocol, IPv6, in PVS environment. They developed theories for networks, IPv6 headers, and routing. They presented the network as an undirected graph. The nodes in the graph represent hosts on the Internet and edges in the graph represent connections between the hosts. They defined a theory called *communication* which defines two predicates, **send?** and **receive?**. These predicates were defined to present the requirements for sending and receiving a packet over a subnet between two hosts. The theory *header* is defined to present the properties of the IPv6 header structure. This header theory is used in this thesis study. Thus, it will be explained in full next. As a last item they defined the theory *routing*. The purpose of the routing theory is to demonstrate several properties of IPv6 routing [40].

3.2 The Header Specification

The theory **header** is the PVS definition of the IPv6 header. To identify the basic IPv6 header and the extension headers, **header_type** was specified by the enumerated type as follows:

```
header_type : TYPE = {IPv6,
                      hop_by_hop,
                      routing,
                      fragment,
                      destination1,
                      destination2,
                      authentication,
                      security,
                      no_next_header,
                      upper_layer}
```

The `next_header` fields of the basic IPv6 header and the extension headers were defined by `TYPE` to identify the type of each header. Extension headers are optional and they are linked similar to a linked list.

All fields in the headers, contain fixed number of bits, in other words each field can hold a subset of integer numbers from 0 to $2^n - 1$, where n is the number of bits in the field. To limit each field between these boundaries the `bitrange` type is defined as:

```
bitrange(n:posnat) : type = subrange(0,(2^n)-1)
```

to create a field of n bits.

Each IPv6 header includes several subfields. For that reason, these headers were defined as PVS records type. As an example, the following specification shows the `fragment_header` definition.

```
fragment_header : type = [# next_header      : header_type,
                           reserved          : bitrange(8),
                           fragment_offset   : bitrange(13),
                           res               : bitrange(2),
                           M_flag           : bool,
                           identification    : nat #]
```

In IPv6, all extension headers are optional. They are appended to the packet if they are needed. Thus, the IPv6 header has a variable structure. However, PVS is not well suited to variant records. While a list is a more suitable structure, the work in [40] used an alternative form. To keep track of the extension headers in a certain packet, the `header_set` type was specified.

```
header_set : type =
  {m : [nat -> header_type] |
   m(0) = IPv6 and
```

```

(forall (i : nat) : m(i) = hop_by_hop => i = 1) and
(forall (i : nat) : m(i) = destination1 =>
  m(i+1) = routing) and
(forall (i,n : nat) : (i < n) &
(m(i) = destination1 =>
  (exists (j : above(0)) : m(i+j) = routing)) &
(m(n) = destination2 =>
  (exists (j : above(0)) : m(n+j) = upper_layer))))}

```

To keep track of the optional extension headers which are appended to a given IPv6 packet, the `header_set` type was specified as an array type.

In order to specify fragmentation and reassembly, `payload_type` was added to the header just for this thesis study. `payload_type` was also added to the specification of the IPv6 `packet_type`.

The basic form of the IPv6 packet, `packet_type`, was defined as a PVS record.

```

packet_type : TYPE =
  [# hs          : header_set,
    IPv6         : IPv6_header,
    hop_by_hop   : hop_by_hop_header,
    routing      : routing_header,
    routing_type_0 : routing_type_0_header,
    fragment     : fragment_header,
    destination1 : destination_options_header,
    destination2 : destination_options_header,
    upper_layer  : upper_layer_header,
    payload      : payload_type #]

```

This packet is a crude form because it includes all the IPv6 headers and payload, and does not include any information about the optionality or order of extension headers. These features were added in the form of the header set, `hs`.

RFC 1883 suggests an order for extension headers. To understand if the extension headers in a given packet are in the suggested order or not, the predicate `order?` was specified.

```

order?(p) : bool =
  forall ( i:nat ) : (
    hs(p)(i) = IPv6          =>
      hs(p)(i+1) = next_header(IPv6(p)) OR
    hs(p)(i) = routing       =>
      hs(p)(i+1) = next_header(routing(p)) OR
    hs(p)(i) = destination1  =>
      hs(p)(i+1) = next_header(destination1(p)) OR
    hs(p)(i) = destination2  =>
      hs(p)(i+1) = next_header(destination2(p)) OR
    hs(p)(i) = hop_by_hop    =>
      hs(p)(i+1) = next_header(hop_by_hop(p)) OR
    hs(p)(i) = fragment      =>
      hs(p)(i+1) = no_next_header OR
    hs(p)(i) = upper_layer   =>
      hs(p)(i+1) = no_next_header OR
    hs(p)(i) = no_next_header =>
      hs(p)(i+1) = no_next_header)

```

If the extension headers in a given packet are in the suggested order, the predicate is **TRUE**, otherwise **FALSE**.

The function `find_header` was defined to detect if a given extension header is appended to a given packet.

```

find_header (p,ht) : bool = exists (i : nat) : (hs(p)(i) = ht)

```

where `p` is a packet and `h` is an extension header type. It returns **TRUE** if the packet includes the given extension header, otherwise, it returns **FALSE**.

Another predicate, `valid_packet?`, was specified in order to add the extension header order feature, by using the `order?` predicate, and other properties of the IPv6 packet.

```

valid_packet?(p) : bool = (
  order?(p)
  AND
  ((find_header(p,hop_by_hop)

```

```

      & option_type(hop_by_hop(p)) = jumbo_payload)
    <=> payload_length(IPv6(p)) = 0)
  AND
  (find_header(p,hop_by_hop)
   & option_type(hop_by_hop(p)) = jumbo_payload
   & jumbo_payload_length(hop_by_hop(p)) >= 2^16
   => not find_header(p,fragment))
  AND
  (find_header(p,routing)
   & routing_type(routing(p)) = routing_type_0
   => not multicast?(destination_address(IPv6(p))))

```

Finally, the IPv6 packet was specified with all properties as a type where `valid_packet?` is TRUE.

```
valid_packet_type : TYPE = ( valid_packet? )
```

This header specification was typechecked and verified. When it is typechecked, PVS generates seven TCCs. Four of these TCCs are subsumed by the other three, and the remaining three are automatically proven. The full specification of this header theory is provided in appendix A.1.

CHAPTER IV

SPECIFICATION OF THE FRAGMENTATION AND REASSEMBLY

The main purpose of this chapter is to demonstrate the specification of the IPv6 packet fragmentation and reassembly in PVS. First, our approach to fragmentation and reassembly is introduced. Then, the type specifications for a packet which does not include the fragment header, and for a packet which includes fragment header is explained. Finally, the specification for the fragmentation and reassembly processes is presented. For verification, all the functions specified for both fragmentation and reassembly combined in the `transmission()` function. This chapter is concluded with the explanation of the `transmission()` function. The specification file for fragmentation and reassemble includes several lemmas and a theorem for verification purposes. These lemmas and the theorem are explained in the next chapter.

4.1 Approach to the Fragmentation and Reassembly

To specify fragmentation and reassembly, several functions are required. Figure 4.1 shows the data flow with function names and the types for the fragmentation process. In the figure, names written in *italic* indicate the type at a given point of the data flow and entities ending with “()” indicate the functions names with parameters as yet unspecified.

During the fragmentation process, the original packet is divided into two parts, a fragmentable part and an unfragmentable part. The unfragmentable part includes

the basic IPv6 header and some extension headers, such as a hop-by-hop header and a routing header if they exist, which must be examined by intermediate nodes in routing the packet. The fragmentable part includes the payload of the original packet and some extension headers that are needed to be processed at the destination.

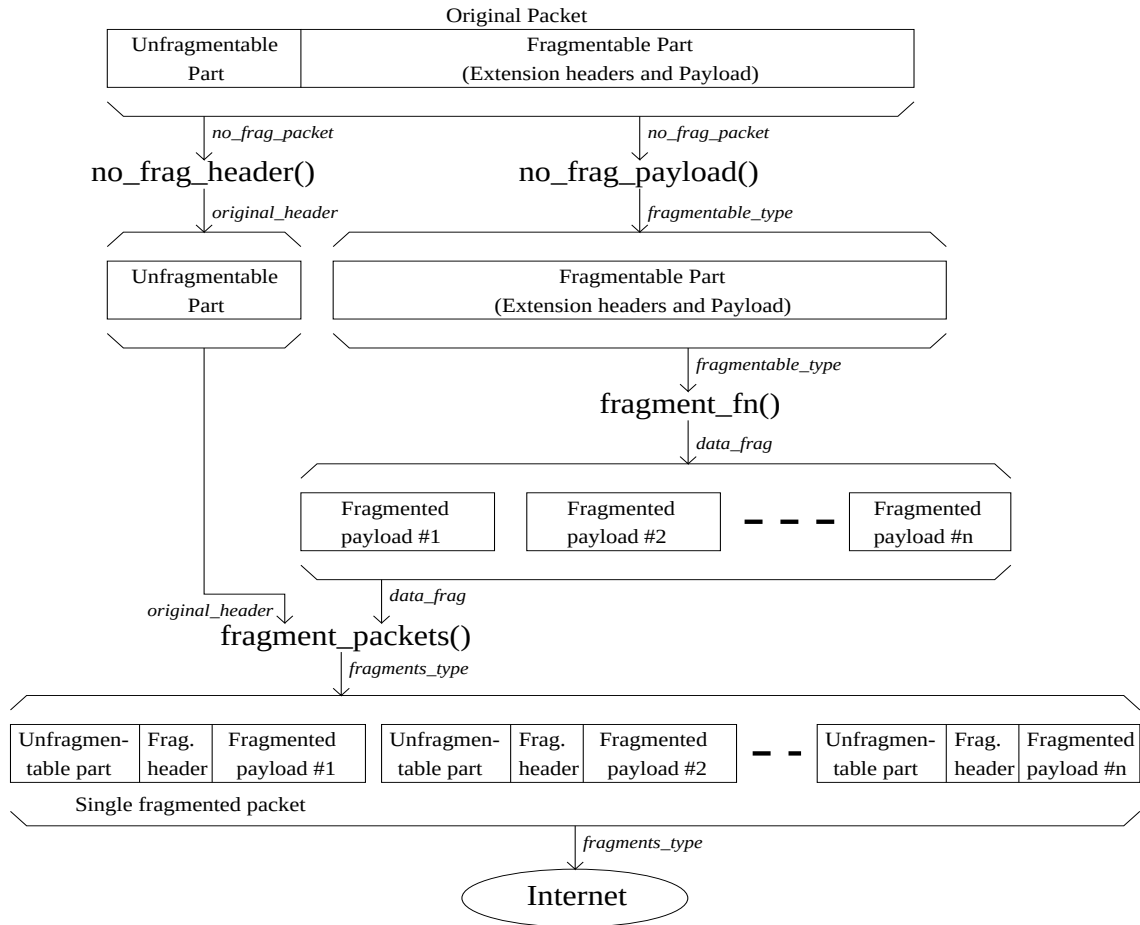


Figure 4.1: Data flow for the fragmentation process.

In the next step of the fragmentation, the fragmentable part is divided into fragments. The last step of the fragmentation process is to assemble each fragment into a packet. To a copy of the unfragmentable part, a fragment header and a

fragment of the fragmentable part are added, and fragmented packet are sent to the Internet. To perform these fragmentation steps, four main functions and a few support functions were specified. The main functions are `no_frag_header()`, `no_frag_payload()`, `fragment_fn()` and `fragment_packets()`

Figure 4.2 shows the data flow through the reassembly process. IPv6 is a connectionless protocol. This means that, it handles each fragment packet individually. For that reason fragment packets may arrive at the receiving host out of order. In the first step, fragment packets are ordered according to the their fragment offset values stored in the offset field of the fragment header. In the second step, the unfragmentable part is captured from the first fragment packet. Then, the unfragmentable parts and fragment headers are removed from each fragment packet and the payloads are kept. These payload are then appended to each other in order to build the original fragmentable part.

After the fragmentable part and the unfragmentable part are captured, these two parts are appended to each other in order to build the original unfragmented packet. At this point, the reassembly process is completed and the packet is transferred to the upper layer protocol.

To perform the reassembly process, five main functions and several support functions were specified. The main functions are `order_fragments()`, `get_header()`, `get_payloads()`, `full_payload()` and `assembly()`.

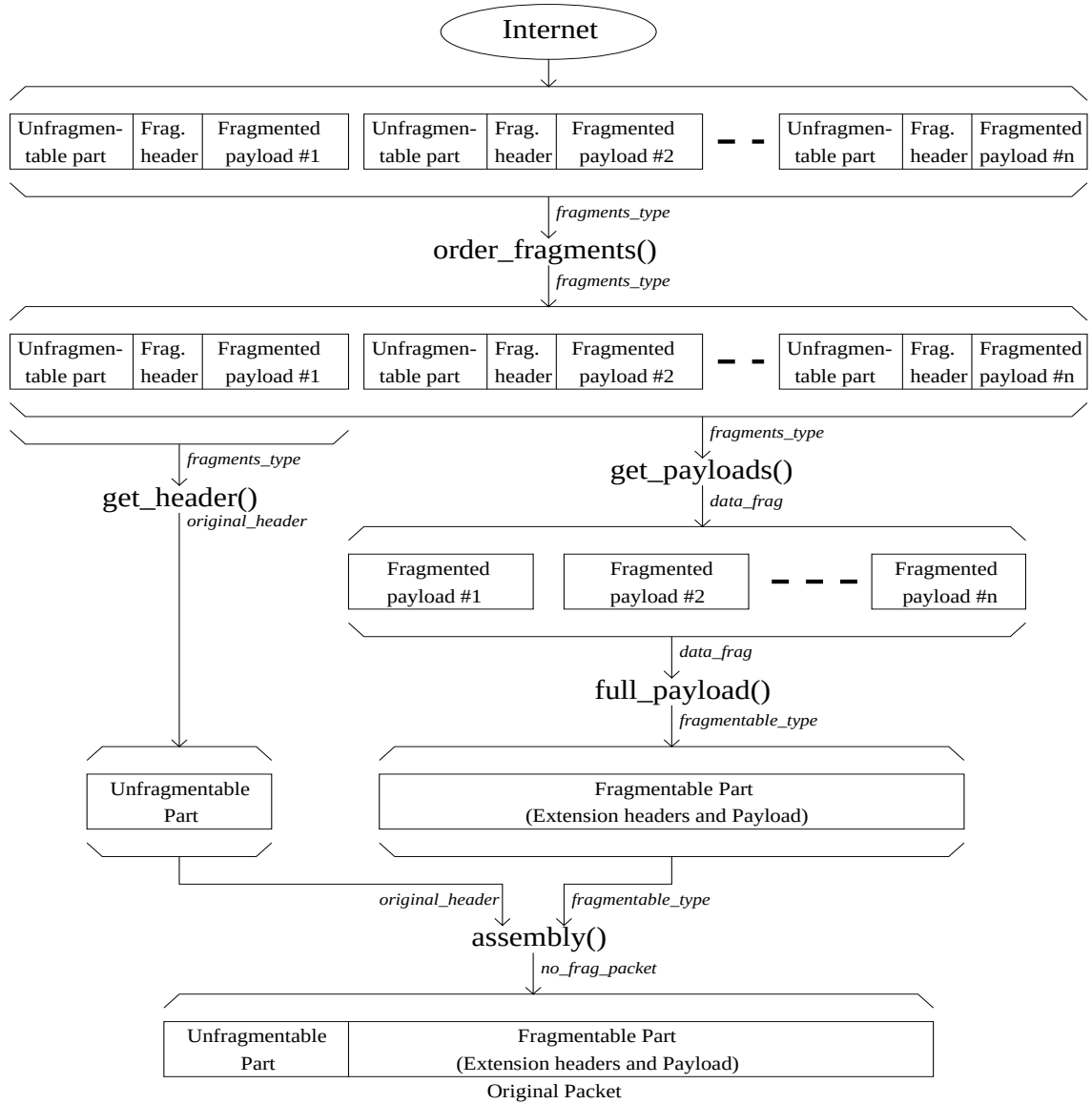


Figure 4.2: Data flow for the reassembly process.

4.2 PVS Specification

The whole fragmentation and reassembly process was specified in one main PVS theory. The type definitions are kept in a separate file. In the following sections, first the type specifications are introduced. After that, parts of the main specification file are presented.

The approach taken in this specification is that the users of the specification defines the function implementations and then passes these implementations to the theory as parameters. The theory **fragmentation** then requires the user to prove certain properties of his functions according to the assumptions of the functions.

The main file begins with the parameter section. In this section MTU and all functions were specified. The functions were specified in the form of **FUNCTION**[$\mathbf{t_1}, \dots, \mathbf{t_n} \rightarrow \mathbf{t}$] where each $\mathbf{t_i}$ is a type expression. MTU was specified as a parameter because its detection is out of this study's scope. After the parameter section, the assuming section begins. In this section, assumptions for some functions were defined. Some of these assumptions identify the implementation of the real functions while some others just identify relations between the functions.

The last section of the main file includes the lemmas and the theorem. These were specified to show that if an implementor of the fragmentation and reassembly procedures of IPv6, implements these functions, the functions will successfully fragment and reassemble the original IPv6 packet, and the reassembled packet will be identical to the original packet.

4.2.1 Type Specifications

For type specifications, the theory **frag_types** is defined. This theory is based on the header specifications in [40] discussed in chapter III. Thus, in this theory the header specification is imported. The purpose of the theory is to specify necessary types such as the original nonfragment packet and the fragment packet for the fragmentation and reassembly of an IPv6 packet. The theory begins with an assuming section. In this section, two assumptions are defined to prove *type-correctness*

conditions (TCCs) generated for the packet definitions. The following PVS code shows these assumptions.

```
no_frag_packet_as: ASSUMPTION
  (EXISTS (x: {v: valid_packet_type |
    (find_header(v, IPv6) AND
    NOT find_header(v, fragment))}):TRUE);

frag_packet_as: ASSUMPTION
  (EXISTS (x: {v: valid_packet_type |
    ((find_header(v, fragment)) AND
    (find_header(v, IPv6)) AND
    (NOT(((find_header(v, destination2)))))) AND
    (NOT(((find_header(v, authentication)))))) AND
    (NOT(((find_header(v, security))))))}): TRUE);
```

While typechecking the theory `frag_types`, PVS generates some existence TCCs for the fragment packet and the original packet specifications. The reason for the existence TCCs is that the fragment packet and the original packet were specified as the `valid_packet_type` with some conditions related to the extension headers. To prove these TCCs, these assumptions were used. The meaning of these assumptions is that the user of this theory should provide these types.

After the assuming section, the first type is defined, `fragmentable_type : TYPE+`. This type was defined to represent an abstraction of the fragmentable part of the IPv6 packet.

The type `no_frag_packet` was specified to represent the original packet which does not include the fragment header. It is specified as a subtype of `valid_packet_type`. An original IPv6 packet must at least include the basic IPv6 header, and must not include the fragment header. To specify this feature of the original packet, two conditions, one for the basic IPv6 header and one for the fragment header were inserted to the `no_frag_packet` specification. No information was provided for the

other extension headers because they are optional and they may or may not appear in the packet. The type `no_frag_packet` is defined as follows:

```
no_frag_packet : TYPE+ = {v: valid_packet_type |
  (find_header(v, IPv6) AND (not find_header(v,fragment)))}
```

For a fragment packet, the type `frag_packet` was specified. A fragment packet must also include the basic IPv6 header. The fragment header must also appear in this header because it is the fragment packet type. For the extension which remain in the unfragmentable part of the packet, no information is provided because they are optional. The extension headers which remain in the fragmentable part, were excluded because they are kept as a part of payload and virtually do not appear in the fragment packets. The following PVS code shows the specification for the `frag_packet` type.

```
frag_packet : TYPE+ = {v: valid_packet_type |
  ((find_header(v, fragment)) AND
  (find_header(v, IPv6)) AND
  (not (find_header(v,destination2))) AND
  (not (find_header(v,authentication))) AND
  (not (find_header(v,security))))}
```

To represent the unfragmentable part of the fragment packet `original_header` was specified. It was defined in the same way with `frag_packet`.

In the rest of the type specification, two intermediate types were specified to represent helping functions' data types. The `id` was specified for the identification field in the fragment header of the fragment packet. After the fragmentable part of the original packet is divided into fragments, to hold fragments, an array type called `data_frag` was specified. The number of items in this array type is identified by `nof_fr_type` which is computed by a support function. The other array type,

`fragments_type`, was specified to hold an assembled fragment packet. The number of items in that array is also identified in the same way with `data_frag`. The figures 4.1 and 4.2 show where these types are used. The specifications of both array types are as follows:

```
data_frag (n:nof_fr_type): TYPE+ = [upto(n) -> payload_type]
fragments_type(n:nof_fr_type) : TYPE+ = [upto(n) -> frag_packet]
```

When this theory is typechecked, PVS generates two existence TCCs, one for `no_frag_packet` and one for `frag_packet` and `original_header`. To prove these TCCs, the assumptions defined in the assuming section of this theory were used.

4.2.2 Fragmentation

To specify the fragmentation process of IPv6, four main functions and several support functions were defined. The main functions are responsible for the data flow through the fragmentation process. All support functions except for `f_fount()` were used in assumptions of the main functions. The `f_count` function was used to identify the number fragments required for a given unfragmented IPv6 packet.

Each function shown in figure 4.1 and support functions are defined below along with the assumptions placed on the function:

- `no_frag_header()`: This function takes an original unfragmented IPv6 packet and gives the unfragmentable part of the packet. It also adds the fragment header to the unfragmentable part with empty fields. Necessary information for the fragment header's fields are assigned in another function. In the assumption of this function, basically the header information of the original packet is assigned to the original header type and the next header values of the extension headers and basic IPv6 header are updated. To figure out the place of the fragment header among all other extension headers, the

`last_header()` function is used. The function `find_header()`, specified in the header theory, is also used to identify the extension headers. The following is the `no_frag_header()` function and its assumption.

```
no_frag_header  : FUNCTION [no_frag_packet -> original_header]
```

```
no_frag_header_assumption : ASSUMPTION
```

```
(FORALL (p1: no_frag_packet): (EXISTS (p2: original_header) :
  hs(p2) = add_frag(p1) &
  IPv6(p2) = IPv6(p1) &
  find_header(p1, hop_by_hop) =>
    (find_header(p2, hop_by_hop) &
     hop_by_hop(p2) = hop_by_hop(p1)) &
  find_header(p1, routing) =>
    (find_header(p2, routing) &
     routing(p2) = routing(p1)) &
  find_header(p1, destination1) =>
    (find_header(p2, destination1) &
     destination1(p2) = destination1(p1)) &
  (IF hs(p1)(last_header(p1)) = IPv6 THEN
    (next_header(fragment(p2))=
      next_header(IPv6(p2)) &
      next_header(IPv6(p2))=fragment)
  ELSIF hs(p1)(last_header(p1)) = routing THEN
    (next_header(fragment(p2))=
      next_header(routing(p2)) &
      next_header(routing(p2))=fragment)
  ELSIF hs(p1)(last_header(p1)) = hop_by_hop THEN
    (next_header(fragment(p2))=
      next_header(hop_by_hop(p2)) &
      next_header(hop_by_hop(p2))=fragment)
  ELSE
    (next_header(fragment(p2))=
      next_header(destination1(p2)) &
      next_header(destination1(p2))=fragment)
  ENDIF)))
```

- `no_frag_payload()`: This function also takes the original packet and returns the fragmentable part of the original packet. The return type of this function

is `fragmentable_type` which is an abstract type and is assumed to include all the extension headers in the fragmentable part and the payload data of the original packet. No assumption is defined for this function. The following PVS code shows the function specification.

```
no_frag_payload: FUNCTION[no_frag_packet -> fragmentable_type]
```

- `fragment_fn()`: After the fragmentable part of the original packet is captured, it must be divided into fragments according to the MTU value of the packet's delivery path. The division process is executed by the `fragment_fn()` function. This function takes two parameters, one is the fragmentable part of the original packet and the other is `nof_fr_type` which is the number of fragments, and is computed by `f_count()`.

```
fragment_fn      : FUNCTION [n:nof_fr_type,  
                             fragmentable_type ->  
                             data_frag(n)]
```

- `fragment_packets()`: After the unfragmentable part and the fragments are created, `fragment_packets()` assembles the fragment packets. This is the last step of the fragmentation process. A copy of the unfragmentable part including the fragment header is appended to each fragment. In this step, fields of each fragment header are also filled. After fragmentation, the `payload_length` field of the basic IPv6 header must be updated. They are updated by this function by calling the `sizeof_header()` and `sizeof_data()` functions. The following PVS code shows both the function and its assumption.

```

fragment_packets: FUNCTION [m:nof_fr_type,
                           original_header,
                           data_frag(m) -> fragments_type(m)]

fragment_packets_assumption : ASSUMPTION
  (FORALL (m:nof_fr_type,h:original_header,d_fr:data_frag(m)):
    (EXISTS (f:fragments_type(m)):
      (FORALL (n:upto(m)):
        (n * sizeof_data(d_fr(0)) <= (213) -1) AND
        (sizeof_header(h)+sizeof_data(d_fr(n)) <= (216)-1) AND
        (IF n<m THEN f(n) = h WITH
          [hs := hs(h),
           IPv6:=IPv6(h)
           WITH [payload_length := sizeof_header(h) +
                sizeof_data(d_fr(n))],
           hop_by_hop:=hop_by_hop(h),
           destination1:=destination1(h),
           routing:=routing(h),
           fragment:= fragment(h)
           WITH [next_header:=next_header(fragment(h)),
                reserved := 0,
                fragment_offset:= n * sizeof_data(d_fr(0)),
                res := 0,
                M_flag := TRUE,
                identification := id ],
           payload:=d_fr(0) ]
        ELSE f(n)=h WITH
          [hs:=hs(h),
           IPv6:=IPv6(h)
           WITH [payload_length := sizeof_header(h) +
                sizeof_data(d_fr(n))],
           hop_by_hop:=hop_by_hop(h),
           destination1:=destination1(h),
           routing:=routing(h),
           fragment:=fragment(h)
           WITH [next_header:=next_header(fragment(h)),
                reserved := 0,
                fragment_offset:= n * sizeof_data(d_fr(0)),
                res := 0,
                M_flag := FALSE,
                identification := id ],
           payload:=d_fr(n) ]
        ENDIF))))

```

The four functions explained above, handle the fragmentation process and use helping functions. In the remaining of this section, support functions for the fragmentation process are introduced.

- **f_count()**: Before dividing the original packet into fragments, IPv6 needs to know how many fragments are required to transmit the packet. The amount of fragments is calculated by **f_count()**. It simply takes the path MTU value and the size of the original packet, and divides the size by MTU. The result value is filtered by the ceiling function, to fix it to an integer number. The **ceiling()** function is defined in **prelude.pvs** which is PVS's build in specification file, and includes some basic definitions. In the assumption of this function, to get the size of the original packet, the **sizeof_packet()** function is called.

```
f_count          : FUNCTION [no_frag_packet -> nof_fr_type]

f_count_assumption : ASSUMPTION
  (FORALL (pk : no_frag_packet) :
    (EXISTS (n : nof_fr_type) :
      n = ceiling(sizeof_packet(pk)/MTU)))
```

- **last_header()**: As mentioned earlier, some of the extension headers must be placed in the unfragmentable part while the others are placed in the fragmentable part. The fragment header must be placed between these two groups of extension headers. In order to find the place of the fragment header, **last_header()** was specified. All extension headers are optional and their occurrence in the packet is identified by the **hs** (header set) field. This function carefully checks the **hs** field and finds the place of the fragment header by finding the last header in the unfragmentable part.

```
last_header      : FUNCTION [no_frag_packet -> nat]
```

```
last_header_assumption : ASSUMPTION
  (FORALL (h1 : no_frag_packet) :
    (EXISTS (i : nat) :
      (((hs(h1)(i) = IPv6) OR
        (hs(h1)(i) = hop_by_hop) OR
        (hs(h1)(i) = destination1) OR
        (hs(h1)(i) = routing)) AND
        ((hs(h1)(i+1) = destination2) OR
          (hs(h1)(i+1) = authentication) OR
          (hs(h1)(i+1) = no_next_header) OR
          (hs(h1)(i+1) = security))))))
```

- `add_frag()`: This function is specified to add a fragment header to the unfragmentable part by updating the `hs` field. By calling `last_header()`, it first identifies the place of the fragment header, and then according to the return value of the function, it appends the fragment header to the unfragmentable part. In fact, the fragment header is added to the unfragmentable part by `no_frag_header()`, but this function cannot perform the append process by itself, rather it calls `add_frag()`.

```
add_frag          : FUNCTION [no_frag_packet -> header_set]
```

```
add_frag_assumption : ASSUMPTION
  (FORALL (h1 : no_frag_packet) :
    (EXISTS (h2 : header_set) :
      (FORALL (i : nat):
        IF (i < last_header(h1) OR (i = last_header(h1))) THEN
          h2(i)=hs(h1)(i)
        ELSIF (i + 1) = last_header(h1) THEN
          h2(i)=fragment
        ELSE
          h2(i+1)=hs(h1)(i)
        ENDIF)))
```

- `sizeof_packet()`: This function is specified to find the total length of the original IPv6 packet including the fragmentable part and the unfragmentable part. This function is called by the `f_count()` function.

`sizeof_packet` : FUNCTION [no_frag_packet -> lof_pk_type]

- `sizeof_frag()`: This function is specified to figure out the length of the fragmentable part of the original packet. In the assumption, it is stated that the length of the fragmentable part is definitely less than the length of the original packet.

`sizeof_frag` : FUNCTION [no_frag_packet -> nat]

`sizeof_frag_type_assumption` : ASSUMPTION
 (FORALL (pk : no_frag_packet) :
 (EXISTS (n : nat) : (n < sizeof_packet(pk))))

- `sizeof_unfrag()`: This function is specified to find the length of the unfragmentable part of the original header before the fragment header is added. In the assumption, it is stated that the size of the original packet is equal to the size of the fragmentable part plus the size of the unfragmentable part.

`sizeof_unfrag` : FUNCTION [no_frag_packet -> posnat]

`sizeof_unfrag_assumption` : ASSUMPTION
 (FORALL (pk : no_frag_packet) : (EXISTS (n : posnat) :
 n + sizeof_frag(pk) = sizeof_packet(pk)))

- `sizeof_header()`: After the fragment packets are generated, the `payload_length` field of the basic IPv6 header must be updated. To update this field,

`fragment_packets` calls the `sizeof_header()` and the `sizeof_data()` functions. `sizeof_header()` simply returns the length of the unfragmentable part after the fragment header is added.

```
sizeof_header    : FUNCTION [original_header -> posnat]
```

- `sizeof_data()`: Like the previous function, the `sizeof_data()` function is also defined to update `payload_length` of the fragment packets. It takes the fragments of the fragmentable part and returns the size of the given fragment.

```
sizeof_data      : FUNCTION [payload_type -> posnat]
```

4.2.3 Reassembly

To specify the reassembly process of IPv6, five main functions and several support functions are defined. Similar to fragmentation, the main functions are responsible for the data flow through the reassembly process. All main functions except for the `order_fragments()` function work as an inverse function of a fragmentation function.

The main functions for the reassembly process shown in figure 4.2 along with their assumptions, can be explained as follows:

- `order_fragments()`: As mentioned earlier, fragment packets may arrive to the final destination address out of order. Before they are reassembled, they must be put in order. The `order_fragments()` function was specified to perform the fragment packet ordering process. This function orders the packets according to the fragment offset value located in the `offset` field of the fragment header of each fragment packet. For this function, two assumptions were

specified. The first assumption states the ordering process. The second assumption states the relation between the `fragment_packet()` function and the `order_fragments()` function.

```

order_fragments : FUNCTION [m:nof_fr_type,
                             fragments_type(m) ->
                             fragments_type(m)]

order_fragments_assumption : ASSUMPTION
  (FORALL (m:nof_fr_type,f:fragments_type(m)):
    (EXISTS (f1:fragments_type(m)):
      (FORALL (i:upto(m)):
        (EXISTS (j:upto(m)):f(i)=f1(j)) &
        (FORALL(k,l:upto(m)):
          k<l => fragment_offset(fragment(f1(k))) <
            fragment_offset(fragment(f1(l))))))))))

order_fragments_asmp : ASSUMPTION
  (FORALL (m:nof_fr_type,f:fragments_type(m),
    h:original_header,d:data_frag(m)):
    (order_fragments (m,fragment_packets(m,h,d)) =
      fragment_packets(m,h,d)))

```

- `get_header()`: During the reassembly process, to get the unfragmentable part of the original packet, `get_header()` was specified. It does the same job as `no_frag_header()` with two differences. The first difference between these functions is their input parameters. `get_header()` takes fragment packets as an input, while `no_frag_header()` takes the original unfragmented packets. The other difference is the process on fragment header. `no_frag_header()` adds the fragment header to the unfragmentable part while `get_header()` removes the fragment header. `rm_frag()` is called in one of the assumptions of the `get_header()` function to remove the fragment header. From another point of view, the function `get_header()` together with the `get_payload()`

function can be considered as an inverse of the `fragment_packet()` function, each for one parameter of `fragment_packet()`. Two assumptions were specified, one for the implementation of this function, and the other is to establish the relationship between this function and the `fragment_packet()` function.

```

get_header      : FUNCTION [n:nof_fr_type,
                           fragments_type(n) ->
                           original_header],

get_header_assumption : ASSUMPTION
  (FORALL (n:nof_fr_type, p1:fragments_type(n)):
    (EXISTS (p2:original_header):
      (new_size(n,p1) <= (2^16)-1) AND
      (hs(p2) = rm_frag(n,p1) AND
       IPv6(p2) = IPv6(p1(0))
       WITH [payload_length :=new_size(n,p1)] AND
       find_header(p1(0), hop_by_hop) =>
         (find_header(p2, hop_by_hop) AND
          hop_by_hop(p2) = hop_by_hop(p1(0))) AND
       find_header(p1(0), routing) =>
         (find_header(p2, routing) AND
          routing(p2) = routing(p1(0))) AND
       find_header(p1(0), destination1) =>
         (find_header(p2, destination1) AND
          destination1(p2) = destination1(p1(0))))))

get_header_asmp : ASSUMPTION
  (FORALL (n:nof_fr_type, p:fragments_type(n),
           h:original_header,d:data_frag(n)):
    (get_header(n,fragment_packets(n,h,d)) = h))

```

- `get_payload()`: Fragmented payloads are taken from the fragment packets by using the `get_payload()` function. This function takes all the fragment packets and returns an array of fragmented payload. It was also specified as an inverse of the `fragment_packet()` function for the `data_frag` parameter of the function. Two assumptions were also specified for the `get_payload()`

function. The first assumption identifies the implementation of the function. The other assumption establishes the relationship between this function and the `fragment_packet()` function.

```
get_payloads      : FUNCTION [m:nof_fr_type,
                             fragments_type(m) -> data_frag(m)],

get_payloads_assumption : ASSUMPTION
  (FORALL (m:nof_fr_type, p:fragments_type(m)):
    (EXISTS (d:data_frag(m)):
      (FORALL(i:upto(m)):d(i)=payload(p(i))))))

get_payloads_asmp : ASSUMPTION
  (FORALL (m:nof_fr_type,h:original_header, d:data_frag(m)):
    (get_payloads(m,fragment_packets(m,h,d))=d))
```

- `full_payload()`: The function `full_payload()` was specified as an inverse of `fragment_fn()`. This function takes fragmented payloads generated by the `get_payloads()` function, and returns the fragmentable part of the original packet. Again, two assumptions were specified for this function. The first assumption implies the implementation of the function and the second assumption puts a relation between this function and the `fragment_fn()` function.

```
full_payload      : FUNCTION [n:nof_fr_type,
                             data_frag(n) -> fragmentable_type],

full_payload_assumption : ASSUMPTION
  (FORALL (n1 : nof_fr_type):
    (EXISTS (d:data_frag(n1)):
      (fragment_fn(n1,full_payload(n1,d)) = d)))

full_payload_asmp : ASSUMPTION
  (FORALL (n:nof_fr_type,d:data_frag(n),f:fragmentable_type):
    (full_payload(n,fragment_fn(n,f)) = f))
```

- **assembly()**: To reassemble the original packet at the destination address, by using the fragmentable part and the unfragmentable part, **assembly()** was specified. It performs the last step of the reassembly process. This function was specified as the exact inverse of the **no_frag_header()** and **no_frag_payload()** functions. For that reason, an implementation assumption was not specified for this function. The only assumption was specified to establish a relation between this function and the **no_frag_header()** and **no_frag_payload()** functions.

```
assembly          : FUNCTION [original_header,
                             fragmentable_type -> no_frag_packet]

assembly_assumption : ASSUMPTION
  (FORALL(pk : no_frag_packet):
    (assembly(no_frag_header(pk),no_frag_payload(pk)) = pk))
```

Besides the main functions, three support functions were specified to remove the fragment header from the unfragmentable part and to update the **payload_length** field of the basic IPv6 header. These functions can be explained as follows.

- **new_size()**: After the original packet is reassembled at the destination address, the **payload_length** field of the basic IPv6 header must be updated in order to reflect the original packet payload length. To do this, **new_size** was specified. This function takes all the fragment packets from the **order_fragments()** function and returns a positive natural number for the payload length of the original reassembled packet. It is called by the **get_header()** function.

```

new_size      : FUNCTION [n:nof_fr_type,
                           fragments_type(n) -> posnat],

```

- `find_frag()`: This function is specified to find the location of the fragment header among the extension headers. It is used by the `rm_frag()` function defined below. It detects the location of the fragment header by simple checking the `hs` (header set) of the first fragment after the fragments are sorted. An assumption was specified for this function.

```

find_frag      : FUNCTION [n:nof_fr_type,
                           fragments_type(n) -> nat],

```

```

find_frag_assumption : ASSUMPTION
  (FORALL (n:nof_fr_type, p:fragments_type(n)):
    (EXISTS (i:nat): hs(p(0))(i) = fragment))

```

- `rm_frag()`: To remove the fragment header from the unfragmentable part of the original packet, `rm_frag()` was specified. This function is called by one of the assumptions of the `get_header()` function. It takes a fragment packet and the number of fragment packets as parameters, and returns a new header set which does not include the fragment header. An assumption was also defined to identify the implementation of the function.

```

rm_frag        : FUNCTION [n:nof_fr_type,
                           fragments_type(n) -> header_set],
rm_frag_assumption : ASSUMPTION
  (FORALL (n:nof_fr_type, p:fragments_type(n)):
    (EXISTS (h:header_set):
      (FORALL (i:nat):
        (IF i < find_frag(n,p) THEN
          h(i) = hs(p(0))(i)
        ELSE
          h(i) = hs(p(0))(i+1)
        ENDIF))))

```

4.2.4 The `transmission()` Function.

In this function, all the main functions and the function `f_count()` were combined in order to simulate the data flow through the fragmentation and reassembly processes. This function takes the `no_frag_packet` type data and returns the same type data. In the function, all the functions call each other in a nested manner. The original packet is first fragmented, and then these fragment packets are fed to the reassembly functions to rebuild the original packet. The `transmission()` function returns this rebuilt original packet. This function is called in the lemmas and in the theorem, in order for simulating whole fragmentation and reassembly processes.

To use this function, two variables, `pk_original` and `pk_transmitted` were specified. The following PVS code is the function `transmission()`.

```
transmission(pk:no_frag_packet):
  no_frag_packet =
    assembly(get_header(f_count(pk),
      order_fragments(f_count(pk),
        fragment_packets(f_count(pk),
          no_frag_header(pk),
          fragment_fn(f_count(pk),
            no_frag_payload(pk))))),
    full_payload(f_count(pk),
      get_payloads(f_count(pk),
        order_fragments(f_count(pk),
          fragment_packets(f_count(pk),
            no_frag_header(pk),
            fragment_fn(f_count(pk),
              no_frag_payload(pk))))))
```

CHAPTER V

VERIFICATION OF THE FRAGMENTATION AND REASSEMBLY PROCESS

In this chapter, verification of the fragmentation and reassembly process is introduced. By verifying the process, it is proven that if a user provides functions meeting the assumptions, then on conclusion of fragmentation and reassembly, he will get back his original information.

5.1 Lemmas for Verification

In the last part of the specification of the fragmentation and reassembly in IPv6, several lemmas and a theorem were specified for verification purposes. The first two lemmas were defined in order to verify the header set of the original IPv6 packet. The following five lemmas were specified for the basic IPv6 header and extension headers. The last lemma was defined for the payload of the original packet.

In the rest of this section, each lemma is explained. In the explanations, the original packet refers to unfragmented original packets and the transmitted packet refers to the packet which is reassembled from the fragment packets of the original packet. Proofs of all lemmas are found in section 5.4.

- **The `hs_1` Lemma:** To keep track of the optional extension headers in the original IPv6 packet, `hs` (header set) was defined. Header set fields of both the original and transmitted packet must include the same information because during the fragmentation and reassembly processes all the extension headers are kept if they appear in the header section of the packet.

```

hs_1 : LEMMA
  pk_transmitted = transmission(pk_original)
  IMPLIES
    hs(pk_transmitted) = hs(pk_original)

```

To verify equality of both packets' header set, the `hs_1` lemma was specified. Although `hs` is included in the IPv6 packet specification, it is not a part of the IPv6 packet, and is just a PVS specification to keep track of the extension headers.

- **The header_set_1 Lemma:** If we find an extension header in the original packets we also must find this header in the transmitted packet.

```

header_set_1 : LEMMA
  (((hs(pk_original) = hs(transmission(pk_original))) AND
    find_header(pk_original, hop_by_hop)) =>
    (find_header(transmission(pk_original), hop_by_hop)))
  AND
  (((hs(pk_original) = hs(transmission(pk_original))) AND
    find_header(pk_original, routing)) =>
    (find_header(transmission(pk_original), routing)))
  AND
  (((hs(pk_original) = hs(transmission(pk_original))) AND
    find_header(pk_original, destination1)) =>
    (find_header(transmission(pk_original), destination1)))
  AND
  (((hs(pk_original) = hs(transmission(pk_original))) AND
    find_header(pk_original, destination2)) =>
    (find_header(transmission(pk_original), destination2))))

```

This is valid for all the extension headers except for the fragment header. The fragment header may only appear in the fragment packet. It is appended to the fragment packets during the fragmentation process and removed during the reassembly process.

- **The Ipv6_l Lemma:** It is an obligation that each IPv6 packet must include at least the basic IPV6 header. The basic IPv6 header in both packets must be the same, except for some fields of the basic IPv6 header such as Hop Limit.

Ipv6_l : LEMMA

```

pk_transmitted = transmission(pk_original)
IMPLIES
(version(IPv6(pk_transmitted)) =
  version(IPv6(pk_original)) AND
priority(IPv6(pk_transmitted)) =
  priority(IPv6(pk_original)) AND
flow_label(IPv6(pk_transmitted)) =
  flow_label(IPv6(pk_original)) AND
payload_length(IPv6(pk_transmitted)) =
  payload_length(IPv6(pk_original)) AND
next_header(IPv6(pk_transmitted)) =
  next_header(IPv6(pk_original)) AND
source_address(IPv6(pk_transmitted)) =
  source_address(IPv6(pk_original)))

```

Some fields of the basic IPv6 header can be changed by hosts through the packets deliver path. For that reason these kind of fields were excluded.

- **The hopbyhop_l Lemma:** If the original packet includes a hop-by-hop options header, the transmitted packet also must include the same header and the content of both headers must be the same.

hopbyhop_l : LEMMA

```

(pk_transmitted = transmission(pk_original) AND
  find_header(pk_original, hop_by_hop))
IMPLIES
(find_header(pk_transmitted, hop_by_hop) AND
  next_header(hop_by_hop(pk_transmitted)) =
    next_header(hop_by_hop(pk_original)) AND
hdr_ext_length(hop_by_hop(pk_transmitted)) =
  hdr_ext_length(hop_by_hop(pk_original)) AND
jumbo_payload_length(hop_by_hop(pk_transmitted)) =
  jumbo_payload_length(hop_by_hop(pk_original)))

```

`hopbyhop_1` was specified to state that both headers are the same. In the hop-by-hop extension header, if the third-highest-order bit of the `option_type` field is set to 1, the option data may be changed by the hosts through the packet's path, so that the `option_data` field is excluded from the lemma.

- **The routing_1 Lemma:** The source of the original packet may choose a specific path for transmission of the packet. In such a case, a routing header is added to the original packet. If a routing header is added to the original packet, it must also appear in the transmitted packet and both routing headers must include the same information in their `routing_type`, `hdr_ext_len` and `next_header` fields.

```
routing_1 : LEMMA
  (pk_transmitted = transmission(pk_original) AND
   find_header(pk_original,routing))
  IMPLIES
  (find_header(pk_transmitted,routing) AND
   next_header(routing(pk_transmitted)) =
    next_header(routing(pk_original)) AND
   hdr_ext_len(routing(pk_transmitted)) =
    hdr_ext_len(routing(pk_original)) AND
   routing_type(routing(pk_transmitted)) =
    routing_type(routing(pk_original)))
```

The other fields of the routing header may be changed by the hosts through the packet's transmission path. Thus, these fields were not included in the lemma.

- **The destination1_1 and destination2_1 Lemmas:** The source of the original packet may add one or more destination options headers to send optional information to the destination . During the fragmentation process, destination options headers can be placed into both the fragmentable and the

unfragmentable part of a packet. If the destination header is placed into the unfragmentable part its options field may be changed during the transmission.

```

destination1_1 : LEMMA
  (pk_transmitted = transmission(pk_original) AND
   find_header(pk_original,destination1))
  IMPLIES
  (find_header(pk_transmitted,destination1) AND
   next_header(destination1(pk_transmitted)) =
    next_header(destination1(pk_original)) AND
   hdr_ext_len(destination1(pk_transmitted)) =
    hdr_ext_len(destination1(pk_original)))

```

In this kind of destination options header, only `next_header` and `hdr_ext_len` must be the same before and after the transmission. On the other hand, a destination options header can be placed into the fragmentable part of a packet. This kind of destination options header becomes a part of the payload until the fragments are reassembled. Thus, all fields of this kind of header must be the same.

```

destination2_1 : LEMMA
  (pk_transmitted = transmission(pk_original) AND
   find_header(pk_original,destination2))
  IMPLIES
  (find_header(pk_transmitted,destination2) AND
   next_header(destination2(pk_transmitted)) =
    next_header(destination2(pk_original)) AND
   hdr_ext_len(destination2(pk_transmitted)) =
    hdr_ext_len(destination2(pk_original)) AND
   options(destination2(pk_transmitted)) =
    options(destination2(pk_original)))

```

- **The payload_1 Lemma:** The payload must be transmitted unchanged even if it is fragmented and reassembled.

```

payload_1 : LEMMA
  pk_transmitted = transmission(pk_original)
  IMPLIES
    payload(pk_transmitted) = payload(pk_original)

```

5.2 The result Theorem

In the result theorem, all the lemmas specified for the headers and payload were combined in order to verify the correctness of the whole fragmentation and reassembly processes. This is included for completeness and is just restatement of the lemmas.

```

result : THEOREM
  ((pk_transmitted = transmission(pk_original)
  IMPLIES
    (version(IPv6(pk_transmitted)) =
      version(IPv6(pk_original)) AND
    priority(IPv6(pk_transmitted)) =
      priority(IPv6(pk_original)) AND
    flow_label(IPv6(pk_transmitted)) =
      flow_label(IPv6(pk_original)) AND
    payload_length(IPv6(pk_transmitted)) =
      payload_length(IPv6(pk_original)) AND
    next_header(IPv6(pk_transmitted)) =
      next_header(IPv6(pk_original)) AND
    source_address(IPv6(pk_transmitted)) =
      source_address(IPv6(pk_original))))
  AND
  ((pk_transmitted = transmission(pk_original) AND
    find_header(pk_original, hop_by_hop))
  IMPLIES
    (find_header(pk_transmitted, hop_by_hop) AND
    next_header(hop_by_hop(pk_transmitted)) =
      next_header(hop_by_hop(pk_original)) AND
    hdr_ext_length(hop_by_hop(pk_transmitted)) =
      hdr_ext_length(hop_by_hop(pk_original)) AND
    jumbo_payload_length(hop_by_hop(pk_transmitted)) =
      jumbo_payload_length(hop_by_hop(pk_original))))
  AND
  ((pk_transmitted = transmission(pk_original) AND

```

```

    find_header(pk_original, routing))
IMPLIES
  (find_header(pk_transmitted, routing) AND
   next_header(routing(pk_transmitted)) =
    next_header(routing(pk_original)) AND
   hdr_ext_len(routing(pk_transmitted)) =
    hdr_ext_len(routing(pk_original)) AND
   routing_type(routing(pk_transmitted)) =
    routing_type(routing(pk_original))))
AND
  ((pk_transmitted = transmission(pk_original) AND
   find_header(pk_original, destination1))
IMPLIES
  (find_header(pk_transmitted, destination1) AND
   next_header(destination1(pk_transmitted)) =
    next_header(destination1(pk_original)) AND
   hdr_ext_len(destination1(pk_transmitted)) =
    hdr_ext_len(destination1(pk_original))))
AND
  ((pk_transmitted = transmission(pk_original) AND
   find_header(pk_original, destination2))
IMPLIES
  (find_header(pk_transmitted, destination2) AND
   next_header(destination2(pk_transmitted)) =
    next_header(destination2(pk_original)) AND
   hdr_ext_len(destination2(pk_transmitted)) =
    hdr_ext_len(destination2(pk_original)) AND
   options(destination2(pk_transmitted)) =
    options(destination2(pk_original))))
AND
  ((pk_transmitted = transmission(pk_original))
IMPLIES
  payload(pk_transmitted) = payload(pk_original)))

```

5.3 Typechecking the Specification

When the whole specification is typechecked, PVS generates 23 *type-correctness conditions* (TCCs). 11 of the total TCCs were subsumed. This means that these TCCs will become TRUE if the other TCCs become TRUE. The remaining 12 TCCs must be proven in order to remove all type obligations. By issuing the PVS proof

checker's `typecheck-prove` command which applies the prover command `subtype-tcc` to each TCC, it is possible to prove 8 of the 12 TCCs. For the final four TCCs, PVS requires assistance from the user. As an example the following TCC is presented.

```
fragment_packets_assumption_TCC4: OBLIGATION
  (FORALL (h: original_header, m: nof_fr_type,
          d_fr: data_frag(m), n: upto(m)):
    (n * sizeof_data(d_fr(0)) <= (2 ^ 13) - 1)
    AND (sizeof_header(h) + sizeof_data(d_fr(n)) <= (2 ^ 16) - 1)
    AND n < m
    IMPLIES 0 <= 0 AND 0 <= (2 ^ 8) - 1);
```

In this TCC, PVS wants us to prove three things. The first one is that `n * sizeof_data(d_fr(0))` is less then or equal to $(2^{13}) - 1$. The second one is `sizeof_header(h) + sizeof_data(d_fr(n))` is less then $(2^{16}) - 1$. The third one is that `n < m` which are variables specified for the number of fragments.

This TCC is generated because of `fragment_packets_assumption`. The conditions that PVS wants us to provide are already defined in this assumption. All we need to do is to help PVS to see them. When we move the cursor to the beginning of this TCC and issue the `prove` command, PVS will respond as follows:

```
fragment_packets_assumption_TCC4 :
|-----
{1}  (FORALL (h: original_header, m: nof_fr_type,
           d_fr: data_frag(m), n: upto(m)):
      (n * sizeof_data(d_fr(0)) <= (2 ^ 13) - 1)
      AND (sizeof_header(h) + sizeof_data(d_fr(n))
           <= (2 ^ 16) - 1)
      AND n < m
      IMPLIES 0 <= 0 AND 0 <= (2 ^ 8) - 1)
```

Rule?

At this point, by issuing the prover command **grind**, it is very easy to prove this TCC. The **grind** strategy is one of the powerful PVS strategies. It first rewrites the definitions and theories and then by applying repeated skolemization, instantiation, and if-lifting it proves the TCCs, lemmas and theorems. This command helps PVS to see previous definitions. When PVS proves this TCC, it gives the following message:

```
Trying repeated skolemization, instantiation, and if-lifting,
Q.E.D.

Run time   = 0.87 secs.
Real time  = 295.34 secs.
NIL
<rcl>
```

The other TCCs are also proved by using the **grind** command.

5.4 Proving the Lemmas and the Theorem

Most of the lemmas specified for verification are proven using the same manner. For that reason, just two lemmas' proofs are introduced. Beside the lemmas, proof of the **result** theorem is also introduced. The PVS proof checker commands that were used to prove the lemmas and the theorem are also briefly explained.

- **Proof of the `hs_1` Lemma:** When the cursor is moved on to this lemma and the **prove** proof checker command is issued in order to initiate proof session for this lemma, PVS responds as follows:

```
hs_1 :
|-----
{1}    (FORALL (pk_original: no_frag_packet,
```

```

        pk_transmitted: no_frag_packet):
    pk_transmitted = transmission(pk_original)
IMPLIES
    hs(pk_transmitted) = hs(pk_original))

```

Rule?

At this point, PVS waits for a proof checker command from the user. In order to prove this lemma, other previously explained assumptions which state relationships between functions, must be used. For this purpose, proof checker's `use` command was used. First `assembly_assumption` was included in the proof. After using (`use "reassembly_assumption"`), PVS responds as follows:

```

Rule? (use "assembly_assumption")
Using lemma assembly_assumption,
this simplifies to:
hs_1 :

{-1}    (FORALL (pk: no_frag_packet):
        (assembly(no_frag_header(pk),
                  no_frag_payload(pk)) = pk))
|-----
[1]    (FORALL (pk_original: no_frag_packet,
              pk_transmitted: no_frag_packet):
        pk_transmitted = transmission(pk_original)
IMPLIES
        hs(pk_transmitted) = hs(pk_original))

```

By using the `use` command, other necessary assumptions were also included to the proof sequent of `hs_1`. These assumptions are `assembly_assumption`, `get_header_asmp`, `full_payload_asmp`, `get_payloads_asmp` and `order_fragments_asmp`. After including the assumptions, PVS's final response is as follows:

```

Rule? (use "order_fragments_asmp")
Using lemma order_fragments_asmp,
this simplifies to:
hs_1 :

{-1}    (FORALL (m: nof_fr_type, f: fragments_type(m),
               h: original_header, d: data_frag(m)):
         (order_fragments(m, fragment_packets(m, h, d)) =
          fragment_packets(m, h, d)))
[-2]    (FORALL (m: nof_fr_type, h: original_header,
               d: data_frag(m)):
         (get_payloads(m, fragment_packets(m, h, d)) = d))
[-3]    (FORALL (n: nof_fr_type, d: data_frag(n),
               f: fragmentable_type):
         (full_payload(n, fragment_fn(n, f)) = f))
[-4]    (FORALL (n: nof_fr_type, p: fragments_type(n),
               h: original_header, d: data_frag(n)):
         (get_header(n, fragment_packets(n, h, d)) = h))
[-5]    (FORALL (pk: no_frag_packet):
         (assembly(no_frag_header(pk),
                   no_frag_payload(pk)) = pk))

|-----
[1]    (FORALL (pk_original: no_frag_packet,
               pk_transmitted: no_frag_packet):
         pk_transmitted = transmission(pk_original)
IMPLIES
         hs(pk_transmitted) = hs(pk_original))

```

Where the antecedents account for the included assumptions and the consequent is the original requirement.

After including the assumptions, the next thing to do is to remove the quantifiers (i.e. `FORALL` or `EXISTS`). In order to remove the quantifier in the consequent, the `SKOSIMP*` command was used. This command removes the quantifier and also flattens the consequent by applying the `flatten` command. When the `flatten` command is applied to a consequent formula of the form $A \supset B$, A becomes an antecedent formula (the one before the `|-----` symbol) and B becomes a consequent formula (the one after the `|-----`

symbol), $(A \vdash B)$. The **SKOSIMP*** command puts the consequent formula [1] into two more tractable pieces. The first part before the **IMPLIES** becomes an antecedent formula and the following part becomes a consequent formula. The following text shows, PVS's response after the **SKOSIMP*** command.

```

Rule? (skosimp*)
Repeatedly Skolemizing and flattening,
this simplifies to:
hs_1 :
[-1]   (FORALL (m: nof_fr_type, f: fragments_type(m),
               h: original_header, d: data_frag(m)):
        (order_fragments(m, fragment_packets(m, h, d)) =
         fragment_packets(m, h, d)))
[-2]   (FORALL (m: nof_fr_type, h: original_header,
               d: data_frag(m)):
        (get_payloads(m, fragment_packets(m, h, d)) = d))
[-3]   (FORALL (n: nof_fr_type, d: data_frag(n),
               f: fragmentable_type):
        (full_payload(n, fragment_fn(n, f)) = f))
[-4]   (FORALL (n: nof_fr_type, p: fragments_type(n),
               h: original_header, d: data_frag(n)):
        (get_header(n, fragment_packets(n, h, d)) = h))
[-5]   (FORALL (pk: no_frag_packet):
        (assembly(no_frag_header(pk),
                  no_frag_payload(pk)) = pk))
{-6}   pk_transmitted!1 = transmission(pk_original!1)

|-----

{1}    hs(pk_transmitted!1) = hs(pk_original!1)

```

At this point, by using PVS's proof strategy **grind**, the sequent was easily proved. The **grind** strategy proves that the original packet's header set and the transmitted packet's header set are equal. The following text is the response of PVS after it proved the lemma.

Rule? (grind)

Trying repeated skolemization, instantiation, and if-lifting,
Q.E.D.

Run time = 17.83 secs.

Real time = 94.82 secs.

NIL

The appearance of Q.E.D implies that the lemma was proven successfully.

- **Proof of the hopbyhop_1 Lemma:** This lemma is proven in a similar manner.

After including all the assumptions and applying the *SKOSIMP** command, the proof sequent looks as follows:

hopbyhop_1 :

```

[-1]   (FORALL (m: nof_fr_type, f: fragments_type(m),
              h: original_header, d: data_frag(m)):
        (order_fragments(m, fragment_packets(m, h, d)) =
         fragment_packets(m, h, d)))
[-2]   (FORALL (m: nof_fr_type, h: original_header,
              d: data_frag(m)):
        (get_payloads(m, fragment_packets(m, h, d)) = d))
[-3]   (FORALL (n: nof_fr_type, d: data_frag(n),
              f: fragmentable_type):
        (full_payload(n, fragment_fn(n, f)) = f))
[-4]   (FORALL (n: nof_fr_type, p: fragments_type(n),
              h: original_header, d: data_frag(n)):
        (get_header(n, fragment_packets(n, h, d)) = h))
[-5]   (FORALL (pk: no_frag_packet):
        (assembly(no_frag_header(pk),
                  no_frag_payload(pk)) = pk))
{-6}   pk_transmitted!1 = transmission(pk_original!1)
{-7}   find_header(pk_original!1, hop_by_hop)
      |-----
{1}   (find_header(pk_transmitted!1, hop_by_hop)
      & next_header(hop_by_hop(pk_transmitted!1))
      = next_header(hop_by_hop(pk_original!1))
      & hdr_ext_length(hop_by_hop(pk_transmitted!1))

```

```

      = hdr_ext_length(hop_by_hop(pk_original!1))
    & jumbo_payload_length(hop_by_hop(pk_transmitted!1))
      = jumbo_payload_length(hop_by_hop(pk_original!1)))

```

The consequent formula (the formula after the |----- symbol) can be divided into smaller pieces to prove easily. The **SPLIT** command divides the formula into 4 subgoals. The following text show the sequent after the **SPLIT** command.

```

Rule? (split)
Splitting conjunctions,
this yields 4 subgoals:
hopbyhop_1.1 :

[-1]   (FORALL (m: nof_fr_type, f: fragments_type(m),
              h: original_header, d: data_frag(m)):
        (order_fragments(m, fragment_packets(m, h, d)) =
          fragment_packets(m, h, d)))
[-2]   (FORALL (m: nof_fr_type, h: original_header,
              d: data_frag(m)):
        (get_payloads(m, fragment_packets(m, h, d)) = d))
[-3]   (FORALL (n: nof_fr_type, d: data_frag(n),
              f: fragmentable_type):
        (full_payload(n, fragment_fn(n, f)) = f))
[-4]   (FORALL (n: nof_fr_type, p: fragments_type(n),
              h: original_header, d: data_frag(n)):
        (get_header(n, fragment_packets(n, h, d)) = h))
[-5]   (FORALL (pk: no_frag_packet):
        (assembly(no_frag_header(pk),
                  no_frag_payload(pk)) = pk))
[-6]   pk_transmitted!1 = transmission(pk_original!1)
[-7]   find_header(pk_original!1, hop_by_hop)
      |-----
{1}    find_header(pk_transmitted!1, hop_by_hop)

```

Rule?

The first of four subgoals is shown. Now, it is very easy to prove these subgoals. The text above shows the first subgoal. To prove this goal we

need to include the assumption `header_set_1` also by invoking the (LEMMA ‘‘`header_set_1`’’) command. After including the lemma, this subgoal was proven by issuing a `GRIND` command. After this subgoal is proven, PVS moves to the second subgoal and responds as follows:

This completes the proof of `hopbyhop_1.1`.

`hopbyhop_1.2` :

```

[-1]   (FORALL (m: nof_fr_type, f: fragments_type(m),
              h: original_header, d: data_frag(m)):
        (order_fragments(m, fragment_packets(m, h, d)) =
          fragment_packets(m, h, d)))
[-2]   (FORALL (m: nof_fr_type, h: original_header,
              d: data_frag(m)):
        (get_payloads(m, fragment_packets(m, h, d)) = d))
[-3]   (FORALL (n: nof_fr_type, d: data_frag(n),
              f: fragmentable_type):
        (full_payload(n, fragment_fn(n, f)) = f))
[-4]   (FORALL (n: nof_fr_type, p: fragments_type(n),
              h: original_header, d: data_frag(n)):
        (get_header(n, fragment_packets(n, h, d)) = h))
[-5]   (FORALL (pk: no_frag_packet):
        (assembly(no_frag_header(pk),
                  no_frag_payload(pk)) = pk))
[-6]   pk_transmitted!1 = transmission(pk_original!1)
[-7]   find_header(pk_original!1, hop_by_hop)
      |-----
{1}    next_header(hop_by_hop(pk_transmitted!1)) =
      next_header(hop_by_hop(pk_original!1))

```

Rule?

The other subgoals were also proven with the `GRIND` command.

The other lemmas are very similar to one of the lemmas whose proof sessions were introduced. Thus, these lemmas were proven by using the method as the `hs_1` or `hop_by_hop_1` lemmas.

- **Proof of the result Theorem:** As was mentioned before, this theorem is a combination of the lemmas; therefore, it is very easy to prove this theorem with lemmas. After removing quantifiers, the theorem is split into subgoals, each corresponding to a lemma. For each subgoal, the corresponding lemma is used and the subgoal is proved. As an example, proof of the first subgoal of the theorem is provided in the following part of this section. The following text shows the first subgoal of the theorem.

```

Splitting conjunctions,
this yields 6 subgoals:
result.1 :
  |-----
{1}      (pk_transmitted!1 = transmission(pk_original!1)
          =>
          (version(IPv6(pk_transmitted!1)) =
            version(IPv6(pk_original!1))
          & priority(IPv6(pk_transmitted!1)) =
            priority(IPv6(pk_original!1))
          & flow_label(IPv6(pk_transmitted!1)) =
            flow_label(IPv6(pk_original!1))
          & payload_length(IPv6(pk_transmitted!1)) =
            payload_length(IPv6(pk_original!1))
          & next_header(IPv6(pk_transmitted!1)) =
            next_header(IPv6(pk_original!1))
          & source_address(IPv6(pk_transmitted!1)) =
            source_address(IPv6(pk_original!1))))

```

Rule?

This subgoal is associated with the `Ipv6` lemma so this lemma was included in the proof sequent. The following text shows the proof sequent after including the lemma.

```

Rule? (lemma "Ipv6_1")
Applying Ipv6_1

```

this simplifies to:
 result.1 :

```
{-1}    (FORALL (pk_original: no_frag_packet,
                pk_transmitted: no_frag_packet):
  pk_transmitted = transmission(pk_original)
  IMPLIES
    (version(IPv6(pk_transmitted)) =
      version(IPv6(pk_original))
    & priority(IPv6(pk_transmitted)) =
      priority(IPv6(pk_original))
    & flow_label(IPv6(pk_transmitted)) =
      flow_label(IPv6(pk_original))
    & payload_length(IPv6(pk_transmitted)) =
      payload_length(IPv6(pk_original))
    & next_header(IPv6(pk_transmitted)) =
      next_header(IPv6(pk_original))
    & source_address(IPv6(pk_transmitted)) =
      source_address(IPv6(pk_original))))
|-----
[1]    (pk_transmitted!1 = transmission(pk_original!1)
      =>
      (version(IPv6(pk_transmitted!1)) =
        version(IPv6(pk_original!1))
      & priority(IPv6(pk_transmitted!1)) =
        priority(IPv6(pk_original!1))
      & flow_label(IPv6(pk_transmitted!1)) =
        flow_label(IPv6(pk_original!1))
      & payload_length(IPv6(pk_transmitted!1)) =
        payload_length(IPv6(pk_original!1))
      & next_header(IPv6(pk_transmitted!1)) =
        next_header(IPv6(pk_original!1))
      & source_address(IPv6(pk_transmitted!1)) =
        source_address(IPv6(pk_original!1))))
```

Now, the antecedent formula and consequent formula look the same except for one difference. If we instantiate the same constants from the consequent formula to the antecedent formula, this will make them equal. If in a sequent, one of the antecedent formulas is equal to one of the consequent formulas, this means that the sequent is *true*. By using, the `INST?` command, it

is possible to instantiate universal constants to the antecedent formula. Right after using the `INST?` command, formulas become equal and the sequent becomes proven. When this subgoal is proven, PVS responds as follows and moves to the next subgoal.

```
Rule? (inst?)
Found substitution:
pk_original gets pk_original!1,
pk_transmitted gets pk_transmitted!1,
Instantiating quantified variables,

This completes the proof of result.1.
```

Using the same method, the other subgoals of the theorem were proven.

The **result** theorem states the correctness of the whole specification. By proving this theorem, we show that all the functions specified for the fragmentation and reassembly process can correctly implement these processes. The message here is that if anyone implements these functions and proves that they meet the assumption requirements, we guarantee that these functions will properly fragment and reassemble the IPv6 packets and the reassembled IPv6 packet will contain the same information as the original unfragmented packet.

CHAPTER VI

CONCLUSIONS

6.1 Results

In this thesis, fragmentation and reassembly in IPv6 were specified using formal methods. As a formal specification environment, the Prototype Verification System (PVS) was used.

Communication protocol specifications are often ambiguous due to providing specifications in the English language. Because the specifications include ambiguity, they are understood differently by different implementers of the communication protocol. On the other hand, a formal specification of a communication protocol does not include ambiguity because it is defined by using mathematical expressions. In this study, by using formal methods, the fragmentation and reassembly process of IPv6 is specified. This formal specification can be used to understand the protocol specification precisely.

Simply specifying a protocol in a formal language does not mean that it is a correct specification. Before both the original specification written in the English language and its associated formal specification are considered correct, the formal specification must be verified. By verifying the formal specification of the fragmentation and reassembly process we showed that our formal specification is correct and by showing that the formal specification is correct, we also showed that the original specification is correct.

By formally specifying the fragmentation and reassembly in IPv6, a formal tool was created for the standardization process of IPv6. A user of this tool, can

pass his function definitions for fragmentation and reassembly as parameters to the specification. The specification requires the user to then prove certain properties of his functions. By proving the specification, it is proven that if the functions passed to the specification meet the requirements of the specifications, then they successfully execute the fragmentation and reassembly process. If a user provides functions meeting the assumptions this means that his function designs are consistent with the standard and he has a working design.

Implementing a communication protocol is very time consuming and costly; therefore, before implementing the protocol, proving its functionality is crucial. While implementing the fragmentation and reassembly process in IPv6, vendors can use the formal tool created in this study to ensure that their design preferences are correct. A goal is for implementers to save time, and thus reduce the cost of their implementations.

6.2 Future Research

This thesis in conjunction with [40] partially specifies the functional requirements of IPv6. Other functionalities still unspecified include:

1. Addressing.
2. Security.
3. Authentication.

Completion of this work should result in an unambiguous specification of IPv6.

Today, a few implementations of IPv6 are available [47, 48]. These implementations were constructed after they were initially designed. A case study can be done about these implementations and the work done in this thesis. In the case study, three topics can be researched. First, whether or not these implementers really want to work with a formal specification instead of a specification written in the English

language. Because of the nature of the formal specification, in some cases, it could be hard to understand some people. As the second topic, whether or not the implementors can easily understand a formal specification can be researched. The goal of the formal specification is to help an implementer save time and reduce the cost of the product. As the third topic, whether or not formal methods can really assist in saving time and reducing the cost of the product can be researched.

REFERENCES

- [1] Sidnie Feit, *TCP/IP, Architecture, Protocols, and Implementation*, McGraw-Hill, Inc., 1993.
- [2] Charles Hedric, "Introduction to the Internet Protocols", <http://oac3.hsc.uth.tmc.edu/staff/snewton/tcp-tutorial/>, (July 17, 1997).
- [3] W. Richard Stevens, *TCP/IP Illustrated Volume 1, The Protocols*, Addison-Wesley Publishing Company, 1994.
- [4] Gray R. Wright and W. Richard Stevens, *TCP/IP Illustrated Volume 2, The Implementation*, Addison-Wesley Publishing Company, 1994.
- [5] Gred E. Keiser, *Local Area Networks*, McGraw-Hill Book Company, 1989.
- [6] Stephen A. Thomas, *IPng and the TCP/IP Protocols*, Wiley Computer Publishing, 1996.
- [7] Scott O. Bradner and A. Mankin, *IPng Internet Protocol Next Generation*, Addison-Wesley Publishing Company, 1996.
- [8] Network Wizards, "Internet Domain Survey", <http://www.nw.com/zone/WWW/top.html>, (July 17, 1997).
- [9] Christian Huitema, *IPv6 The New Internet Protocol*, Prentice Hall PTR, 1996.
- [10] Jon Postel, *Internet Protocol, Darpa Internet Program Protocol Specification*, RFC-791, Sep. 1981.
- [11] David D. Clark *IP Datagram Reassembly Algorithms*, RFC-815, July. 1982.
- [12] T. Socolofsky, C. Kale, *A TCP/IP Tutorial*, RFC-1180, Jan. 1991.
- [13] J. Mogul, S. Deering, *Path MTU Discovery*, RFC-1191, Nov. 1990.
- [14] S. Bradner and A. Mankin, *The Recommendation for the IP Next Generation Protocol*, RFC-1752, Jan. 1995.

- [15] S. Deering, R. Hinden, *IPv6 Specification*, RFC-1883, Dec. 1995.
- [16] J. McCann, S. Deering and J. Mogul, *Path MTU Discovery for IP Version 6*, RFC-1981, Aug. 1996.
- [17] A. Conta and S. Deering, *Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6)*, RFC-1885, Dec. 1995.
- [18] Daniel T. Harrington, James P. Bound, John J. McCann and Matt Thomas, "Internet Protocol Version 6 and the Digital UNIX Implementation Experience", *Digital Technical Journal*, vol. 8, no. 3, pp. 5-21, 1996.
- [19] William Stallings, "IPv6: The New Internet Protocol", *IEEE Communications Magazine*, vol. 34, no. 7, pp. 96-108, Jul. 1996.
- [20] Robert L. Popp, "Implications of Internet Fragmentation and Transit Network Authentication", *Proceedings of the First International Conference on Local Area Network Interconnection*, pp. 209 - 31, October 1993.
- [21] M. A. Bonuccelli, "Minimum Fragmentation Internet Routing", *IEEE INFOCOM '91. The Conference on Computer Communications. Proceedings. Tenth Annual Joint Conference of the IEEE*, pp. 289-94, April 1991.
- [22] Robert M. Hiden, "IP Next Generation Overview", *Communications of The ACM*, vol. 39, no. 6, pp. 62-71, June 1996.
- [23] J. F. Shoch, "Packet fragmentation in inter-network protocols", *Computer Networks*, vol. 6, 1979, pp. 3-8.
- [24] C. A. Sunshine, "Network interconnection and gateways", *IEEE Journal on Selected Areas in Communications*, vol. 8, 1990, pp. 4-11.
- [25] D. L. Mills, P. Schragger, and M. Davies, "Internet architecture workshop: Future of the internet system architecture and TCP/IP protocols", *ACM Computer Communication Review*, January 1990.

- [26] D. Comer, *Internetworking with TCP/IP Volume I*, Prentice Hall, Englewood, NJ, 1991.
- [27] Micheal Hinckey, “Formal Methods”, <http://www.csis.ul.ie/hinckeym/ap/fm.html>, (28 May 1997).
- [28] S. Mauw, G. J. Veltink, *Algebraic Specification of Communication Protocols*, Cambridge University Press, 1993.
- [29] John Rushby, “Mechanized Formal Methods: Progress and Prospects”, *16th Conference on the Foundations of Software Technology and Theoretical Computer Science*, Hyderabad, India, December 1996.
- [30] Jonathan Bowen, *Formal Specification and Documentation Using Z*, International Thomson Computer Press, 1996.
- [31] Constance Heitmeyer and Dino Mandrioli, *Formal Methods for Real Time Computing*, John Wiley and Sons Inc., 1996.
- [32] S. Owre, S. Rajan, J. M. Rushby, N. Shankar and M. Srivas, “PVS: Combining Specification, Proof Checking, and Model Checking”, *Computer Aided Verification, 8th International Conference*, July 1996.
- [33] J. Crow, S. Owre, J. Rushby, N. Shankar and M. Srivas “A Tutorial Introduction to PVS”, *Computer Science Laboratory, SRI International*, June 1995.
- [34] S. Owre, N. Shankar and J. M. Rushby, “User Guide for the PVS Specification and Verification System”, *Computer Science Laboratory, SRI International*, March 1993.
- [35] Daniel E. Atkins, “Higher-Radix Division Using Estimates of the Divisor and Partial Remainder”, *Transactions on Computers*, vol. C-17, no. 10, pp. 925-34 October 1968.
- [36] S. Owre, N. Shankar and J. M. Rushby, “The PVS Specification Language”, *Computer Science Laboratory, SRI International*, March 1993.

- [37] S. Owre, N. Shankar and J. M. Rushby, “The PVS Proof Checker: A Reference Manual”, *Computer Science Laboratory, SRI International*, March 1993.
- [38] Tat Y. Choi, “Formal Techniques for the Specification, Verification, and construction of Communication Protocols”, *IEEE Communication Magazine*, vol. 23, no. 10, pp. 46-52, October 1985.
- [39] James K. Huggins, “Kermit: Specification and Verification”, <http://kwaziwai.cc.columbia.edu/kermit/proof.html>, (22 July 1997)
- [40] J. Leathrum, R. Morsi, and T. Leathrum, “Formal Verification of Communication Protocols,” *IASTED Int. Conf. on Parallel and Distributed Systems*, Oct. 1996.
- [41] A. J. Galloway “The Vienna Development Method”, <http://www-scm.tees.ac.uk/bresource/docs/Dave.ZVdmB/node6.html> (12 August 1997).
- [42] T. Bolognesi and E. Brinksma, “Introduction to the ISO Specification Language LOTOS”, *Computer Networks and ISDN Systems 14*, pp. 25-59, Elsevier Science Publishing, 1987.
- [43] F. Belina, D. Hogrefe and A. Sarma *SDL with Applications from Protocol Specification*, Prentice Hall International (UK) Ltd., 1991.
- [44] H. Ruess, M. K. Srivas, and N. Shankar., “Modular Verification of SRT division”, *Computer-Aided Verification, CAV’96*, Lecture Notes in Computer Science, New Brunswick, NJ, July 1996.
- [45] Paul, S. Miner and James F. Leathrum, Jr., “Verification of IEEE Compliant Subtractive Division Algorithms”, *To appear in FMCAD ’96*.
- [46] Milica Barjaktarovic, “Formal Specification and Verification of the OSI Session Layer Using the Calculus of Communicating Systems”, *Ph.D. Thesis, Department of Electrical and Computer Engineering, Syracuse University*, August 1995.

- [47] R. J. Atkinson, D.L. McDonald, B. G. Phan, C. W. Metz and K. C. Chin,
“Implementation of IPv6 in 4.4 BSD”, *1996 USENIX Technical Conference*,
January 22-26, 1996, San Diego, CA.
- [48] A. Schill, Sabine Kuhn and F. Breiter, “Internetworking Over ATM: Experience with IP/IPng and RSVP”, *Computer Networks and ISDN Systems*, v. 28
pp. 1915-27, July 1996.

APPENDICES

THE SPECIFICATION AND PROOF FILES

A The Specification Files

A.1 Specification of the IPv6 Header

```

header : THEORY
begin

importing network

payload_type : TYPE+

header_type : TYPE = {IPv6, hop_by_hop, routing,
                      fragment, destination1,
                      destination2, authentication,
                      security, no_next_header,
                      upper_layer}

bitrange(n : posnat) : type = subrange(0, (2^n)-1)

IPv6_header : TYPE = [# version           : nat,
                      priority             : bitrange(4),
                      flow_label           : bitrange(24),
                      payload_length       : bitrange(16),
                      next_header          : header_type,
                      hop_limit            : bitrange(8),
                      source_address       : address,
                      destination_address  : address #]

upper_layer_header : type+

hop_by_hop_type : type = {jumbo_payload, Pad1, PadN}

hop_by_hop_header : type =
  [# next_header      : header_type,
   hdr_ext_length     : bitrange(8),
   option_type        : hop_by_hop_type,
   jumbo_payload_length : bitrange(32) #]

```

```

routing_type : type = {routing_type_0}
routing_header : type =
  [# next_header      : header_type,
   hdr_ext_len        : {n : nat | n <= 46 & even?(n)},
   routing_type       : routing_type,
   segments_left      : subrange(0,hdr_ext_len/2),
   type_specific_data : nat #]

routing_type_0_header : type =
  [# reserved : bitrange(8),
   strict     : [subrange(0,23) -> bool],
   address    : [subrange(1,23) -> non_multi_addr] #]

fragment_header : type =
  [# next_header      : header_type,
   reserved           : bitrange(8),
   fragment_offset    : bitrange(13),
   res                : bitrange(2),
   M_flag             : bool,
   identification     : nat #]

destination_options_header : type =
  [# next_header : header_type,
   hdr_ext_len  : bitrange(8),
   options      : nat #]

header_set : type =
  {m : [nat -> header_type] |
   m(0) = IPv6 and
   (forall (i : nat) : m(i) = hop_by_hop => i = 1) and
   (forall (i : nat) : m(i) = destination1 =>
    m(i+1) = routing) and
   (forall (i,n : nat) : (i < n) &
    (m(i) = destination1 =>
     (exists (j : above(0)) : m(i+j) = routing)) &
    (m(n) = destination2 =>
     (exists (j : above(0)) : m(n+j) = upper_layer))))}

packet_type : TYPE =
  [# hs      : header_set,
   IPv6      : IPv6_header,
   hop_by_hop : hop_by_hop_header,
   routing    : routing_header,
   routing_type_0 : routing_type_0_header,
   fragment   : fragment_header,

```

```

    destination1    : destination_options_header,
    destination2    : destination_options_header,
    upper_layer     : upper_layer_header,
    payload         : payload_type #]

p: var packet_type

ht: var header_type

order?(p) : bool =
  forall ( i:nat ) : (
    hs(p)(i) = IPv6 =>
      hs(p)(i+1) = next_header(IPv6(p)) OR
    hs(p)(i) = routing =>
      hs(p)(i+1) = next_header(routing(p)) OR
    hs(p)(i) = destination1 =>
      hs(p)(i+1) = next_header(destination1(p)) OR
    hs(p)(i) = destination2 =>
      hs(p)(i+1) = next_header(destination2(p)) OR
    hs(p)(i) = hop_by_hop =>
      hs(p)(i+1) = next_header(hop_by_hop(p)) OR
    hs(p)(i) = fragment =>
      hs(p)(i+1) = no_next_header OR
    hs(p)(i) = upper_layer =>
      hs(p)(i+1) = no_next_header OR
    hs(p)(i) = no_next_header =>
      hs(p)(i+1) = no_next_header)

find_header (p,ht) : bool = exists (i : nat) : (hs(p)(i) = ht)

valid_packet?(p) : bool = (
  order?(p)
  AND
  ((find_header(p,hop_by_hop)
    & option_type(hop_by_hop(p)) = jumbo_payload)
  <=>  payload_length(IPv6(p)) = 0)
  AND
  (find_header(p,hop_by_hop)
    & option_type(hop_by_hop(p)) = jumbo_payload
    & jumbo_payload_length(hop_by_hop(p)) >= 2^16
    => not find_header(p,fragment))
  AND
  (find_header(p,routing)
    & routing_type(routing(p)) = routing_type_0
    % => destination_address(IPv6(p)) = (non_multi_addr))

```

```

=> not multicast?(destination_address(IPv6(p))))
valid_packet_type : TYPE = (valid_packet?)

end header

```

A.2 Type Specifications

```

frag_types : THEORY

BEGIN

ASSUMING

importing header

no_frag_packet_as : ASSUMPTION
  (EXISTS (x : {v : valid_packet_type |
    (find_header(v, IPv6) AND
    NOT find_header(v, fragment))}) : TRUE);

frag_packet_as : ASSUMPTION
  (EXISTS (x : {v : valid_packet_type |
    ((find_header(v, fragment)) AND
    (find_header(v, IPv6)) AND
    (NOT(((find_header(v, destination2)))) AND
    (NOT(((find_header(v, authentication)))) AND
    (NOT(((find_header(v, security))))))}) : TRUE);

ENDASSUMING

importing header

fragmentable_type : TYPE+

no_frag_packet : TYPE+ = {v : valid_packet_type |
  (find_header(v, IPv6) and
  (not find_header(v,fragment)))}

frag_packet : TYPE+ = {v : valid_packet_type |
  ((find_header(v, fragment)) AND
  (find_header(v, IPv6)) AND
  (not (find_header(v,destination2))) AND
  (not (find_header(v,authentication))) AND
  (not (find_header(v,security))))}

```

```

original_header : TYPE+ = {v: valid_packet_type |
  ((find_header(v, fragment)) AND
   (find_header(v, IPv6)) AND
   (not (find_header(v, destination2))) AND
   (not (find_header(v, authentication))) AND
   (not (find_header(v, security))))}

nof_fr_type      : TYPE+ = nat
lof_pk_type      : TYPE+ = posnat
id               : nat

data_frag (n : nof_fr_type) : TYPE+ = [upto(n) -> payload_type]

fragments_type(n : nof_fr_type) : TYPE+ = [upto(n) -> frag_packet]

END frag_types

```

A.3 Specification of the Fragmentation and Reassembly

```

fragmentation [ MTU : posnat,
  (importing header, frag_types)
  no_frag_payload : FUNCTION [no_frag_packet -> fragmentable_type],
  last_header     : FUNCTION [no_frag_packet -> nat],
  add_frag        : FUNCTION [no_frag_packet -> header_set],
  no_frag_header  : FUNCTION [no_frag_packet -> original_header],
  sizeof_packet   : FUNCTION [no_frag_packet -> lof_pk_type],
  sizeof_frag     : FUNCTION [no_frag_packet -> nat],
  sizeof_unfrag   : FUNCTION [no_frag_packet -> posnat],
  f_count         : FUNCTION [no_frag_packet -> nof_fr_type],
  fragment_fn     : FUNCTION [n:nof_fr_type,
                              fragmentable_type -> data_frag(n)],
  sizeof_header   : FUNCTION [original_header -> posnat],
  sizeof_data     : FUNCTION [payload_type -> posnat],
  fragment_packets: FUNCTION [m:nof_fr_type,
                              original_header,
                              data_frag(m) -> fragments_type(m)],
  order_fragments : FUNCTION [m:nof_fr_type,
                              fragments_type(m) ->
                              fragments_type(m)],
  get_payloads    : FUNCTION [m:nof_fr_type,
                              fragments_type(m) -> data_frag(m)],
  full_payload    : FUNCTION [n:nof_fr_type,
                              data_frag(n) -> fragmentable_type],

```

```

new_size      : FUNCTION [n:nof_fr_type,
                          fragments_type(n) -> posnat],
find_frag     : FUNCTION [n:nof_fr_type,
                          fragments_type(n) -> nat],
rm_frag       : FUNCTION [n:nof_fr_type,
                          fragments_type(n) -> header_set],
get_header    : FUNCTION [n:nof_fr_type,
                          fragments_type(n) ->
                          original_header],
assembly      : FUNCTION [original_header,
                          fragmentable_type -> no_frag_packet]
] : THEORY

BEGIN

ASSUMING

last_header_assumption : ASSUMPTION
  (FORALL (h1 : no_frag_packet) : (EXISTS (i : nat) :
    (((hs(h1)(i) = IPv6) OR
     (hs(h1)(i) = hop_by_hop) OR
     (hs(h1)(i) = destination1) OR
     (hs(h1)(i) = routing)) AND
    ((hs(h1)(i+1) = destination2) OR
     (hs(h1)(i+1) = authentication) OR
     (hs(h1)(i+1) = no_next_header) OR
     (hs(h1)(i+1) = security))))))

add_frag_assumption : ASSUMPTION
  (FORALL (h1 : no_frag_packet) : (EXISTS (h2 : header_set) :
    (FORALL (i : nat) :
      IF (i < last_header(h1) OR (i = last_header(h1))) THEN
        h2(i)=hs(h1)(i)
      ELSIF (i + 1) = last_header(h1) THEN
        h2(i)=fragment
      ELSE
        h2(i+1)=hs(h1)(i)
      ENDIF)))

no_frag_header_assumption : ASSUMPTION
  (FORALL (p1 : no_frag_packet) : (EXISTS (p2 : original_header) :
    hs(p2) = add_frag(p1) &
    IPv6(p2) = IPv6(p1) &
    find_header(p1, hop_by_hop) =>
      (find_header(p2, hop_by_hop) &

```

```

        hop_by_hop(p2) = hop_by_hop(p1)) &
find_header(p1, routing) =>
    (find_header(p2, routing) &
    routing(p2) = routing(p1)) &
find_header(p1, destination1) =>
    (find_header(p2, destination1) &
    destination1(p2) = destination1(p1)) &
(IF hs(p1)(last_header(p1)) = IPv6 THEN
    (next_header(fragment(p2))=next_header(IPv6(p2)) &
    next_header(IPv6(p2))=fragment)
ELSIF hs(p1)(last_header(p1)) = routing THEN
    (next_header(fragment(p2))=next_header(routing(p2)) &
    next_header(routing(p2))=fragment)
ELSIF hs(p1)(last_header(p1)) = hop_by_hop THEN
    (next_header(fragment(p2))=next_header(hop_by_hop(p2)) &
    next_header(hop_by_hop(p2))=fragment)
ELSE
    (next_header(fragment(p2))=next_header(destination1(p2)) &
    next_header(destination1(p2))=fragment)
ENDIF)))

sizeof_frag_type_assumption : ASSUMPTION
(FORALL (pk : no_frag_packet) :
    (EXISTS (n : nat) : (n < sizeof_packet(pk))))

sizeof_unfrag_assumption : ASSUMPTION
(FORALL (pk : no_frag_packet) : (EXISTS (n : posnat) :
    n + sizeof_frag(pk) = sizeof_packet(pk)))

f_count_assumption : ASSUMPTION
(FORALL (pk : no_frag_packet) :
    (EXISTS (n : nof_fr_type) :
        n = ceiling(sizeof_packet(pk)/MTU)))

fragment_packets_assumption : ASSUMPTION
(FORALL (m:nof_fr_type, h:original_header, d_fr:data_frag(m)):
    (EXISTS (f:fragments_type(m)):
        (FORALL (n:upto(m)):
            (n * sizeof_data(d_fr(0)) <= (2^13) -1) AND
            (sizeof_header(h) + sizeof_data(d_fr(n)) <= (2^16)-1) AND
            (IF n<m THEN f(n) = h WITH
                [hs := hs(h),
                IPv6:=IPv6(h)
                WITH [payload_length := sizeof_header(h) +
                    sizeof_data(d_fr(n))],

```

```

hop_by_hop:=hop_by_hop(h),
destination1:=destination1(h),
routing:=routing(h),
fragment:= fragment(h)
  WITH [next_header:=next_header(fragment(h)),
        reserved := 0,
        fragment_offset:= n * sizeof_data(d_fr(0)),
        res := 0,
        M_flag := TRUE,
        identification := id ],
payload:=d_fr(0) ]
ELSE f(n)=h WITH
[hs:=hs(h),
IPv6:=IPv6(h)
  WITH [payload_length := sizeof_header(h) +
        sizeof_data(d_fr(n))],

hop_by_hop:=hop_by_hop(h),
destination1:=destination1(h),
routing:=routing(h),
fragment:=fragment(h)
  WITH [next_header:=next_header(fragment(h)),
        reserved := 0,
        fragment_offset := n * sizeof_data(d_fr(0)),
        res := 0,
        M_flag := FALSE,
        identification := id ],
payload:=d_fr(n) ]
ENDIF))))

```

```

order_fragments_assumption : ASSUMPTION
(FORALL (m:nof_fr_type,f:fragments_type(m)):
(EXISTS (f1:fragments_type(m)):
(FORALL (i:upto(m)):
(EXISTS (j:upto(m)):f(i)=f1(j)) &
(FORALL(k,l:upto(m)):
k<l => fragment_offset(fragment(f1(k))) <
fragment_offset(fragment(f1(l)))))))

```

```

order_fragments_asmp : ASSUMPTION
(FORALL (m:nof_fr_type,f:fragments_type(m),
h:original_header,d:data_frag(m)):
(order_fragments (m,fragment_packets(m,h,d)) =
fragment_packets(m,h,d)))

```

```

get_payloads_assumption : ASSUMPTION
  (FORALL (m:nof_fr_type, p:fragments_type(m)):
    (EXISTS (d:data_frag(m)):
      (FORALL(i:upto(m)):d(i)=payload(p(i))))))

get_payloads_asmp : ASSUMPTION
  (FORALL (m:nof_fr_type,h:original_header, d:data_frag(m)):
    (get_payloads(m,fragment_packets(m,h,d))=d))

full_payload_assumption : ASSUMPTION
  (FORALL (n1 : nof_fr_type):
    (EXISTS (d:data_frag(n1)):
      (fragment_fn(n1,full_payload(n1,d)) = d)))

full_payload_asmp : ASSUMPTION
  (FORALL (n:nof_fr_type,d:data_frag(n),f:fragmentable_type):
    (full_payload(n,fragment_fn(n,f)) = f))

find_frag_assumption : ASSUMPTION
  (FORALL (n:nof_fr_type, p:fragments_type(n)):
    (EXISTS (i:nat): hs(p(0))(i) = fragment))

rm_frag_assumption : ASSUMPTION
  (FORALL (n:nof_fr_type, p:fragments_type(n)):
    (EXISTS (h:header_set):
      (FORALL (i:nat):
        (IF i < find_frag(n,p) THEN
          h(i) = hs(p(0))(i)
        ELSE
          h(i) = hs(p(0))(i+1)
        ENDIF))))))

get_header_assumption : ASSUMPTION
  (FORALL (n:nof_fr_type, p1:fragments_type(n)):
    (EXISTS (p2:original_header):
      (new_size(n,p1) <= (216)-1) AND
      (hs(p2) = rm_frag(n,p1) AND
      IPv6(p2) = IPv6(p1(0))
      WITH [payload_length :=new_size(n,p1)] AND
      find_header(p1(0), hop_by_hop) =>
      (find_header(p2, hop_by_hop) AND
      hop_by_hop(p2) = hop_by_hop(p1(0))) AND
      find_header(p1(0), routing) =>
      (find_header(p2, routing) AND
      routing(p2) = routing(p1(0))) AND

```

```

    find_header(p1(0), destination1) =>
      (find_header(p2, destination1) AND
       destination1(p2) = destination1(p1(0))))))

get_header_asmp : ASSUMPTION
  (FORALL (n:nof_fr_type, p:fragments_type(n),
           h:original_header,d:data_frag(n)):
    (get_header(n,fragment_packets(n,h,d)) = h))

assembly_assumption : ASSUMPTION
  (FORALL(pk : no_frag_packet):
    (assembly(no_frag_header(pk),no_frag_payload(pk)) = pk))

ENDASSUMING

transmission(pk:no_frag_packet): no_frag_packet =
  assembly(get_header(f_count(pk),
    order_fragments(f_count(pk),
      fragment_packets(f_count(pk),
        no_frag_header(pk),
        fragment_fn(f_count(pk),
          no_frag_payload(pk))))),
    full_payload(f_count(pk),
      get_payloads(f_count(pk),
        order_fragments(f_count(pk),
          fragment_packets(f_count(pk),
            no_frag_header(pk),
            fragment_fn(f_count(pk),
              no_frag_payload(pk)))))))

pk_original, pk_transmitted : VAR no_frag_packet

hs_1 : LEMMA
  pk_transmitted = transmission(pk_original)
  IMPLIES
    hs(pk_transmitted) = hs(pk_original)

header_set_1 : LEMMA
  (((hs(pk_original) = hs(transmission(pk_original))) AND
    find_header(pk_original, hop_by_hop)) =>
    (find_header(transmission(pk_original), hop_by_hop))) AND
  (((hs(pk_original) = hs(transmission(pk_original))) AND
    find_header(pk_original, routing)) =>
    (find_header(transmission(pk_original), routing))) AND
  (((hs(pk_original) = hs(transmission(pk_original))) AND

```

```

    find_header(pk_original, destination1)) =>
      (find_header(transmission(pk_original), destination1))) AND
    (((hs(pk_original) = hs(transmission(pk_original))) AND
      find_header(pk_original, destination2)) =>
      (find_header(transmission(pk_original), destination2))))

```

Ipv6_1 : LEMMA

```

  pk_transmitted = transmission(pk_original)
  IMPLIES
    (version(IPv6(pk_transmitted)) =
      version(IPv6(pk_original)) AND
    priority(IPv6(pk_transmitted)) =
      priority(IPv6(pk_original)) AND
    flow_label(IPv6(pk_transmitted)) =
      flow_label(IPv6(pk_original)) AND
    payload_length(IPv6(pk_transmitted)) =
      payload_length(IPv6(pk_original)) AND
    next_header(IPv6(pk_transmitted)) =
      next_header(IPv6(pk_original)) AND
    source_address(IPv6(pk_transmitted)) =
      source_address(IPv6(pk_original)))

```

hopbyhop_1 : LEMMA

```

  (pk_transmitted = transmission(pk_original) AND
  find_header(pk_original, hop_by_hop))
  IMPLIES
    (find_header(pk_transmitted, hop_by_hop) AND
    next_header(hop_by_hop(pk_transmitted)) =
      next_header(hop_by_hop(pk_original)) AND
    hdr_ext_length(hop_by_hop(pk_transmitted)) =
      hdr_ext_length(hop_by_hop(pk_original)) AND
    jumbo_payload_length(hop_by_hop(pk_transmitted)) =
      jumbo_payload_length(hop_by_hop(pk_original)))

```

routing_1 : LEMMA

```

  (pk_transmitted = transmission(pk_original) AND
  find_header(pk_original, routing))
  IMPLIES
    (find_header(pk_transmitted, routing) AND
    next_header(routing(pk_transmitted)) =
      next_header(routing(pk_original)) AND
    hdr_ext_len(routing(pk_transmitted)) =
      hdr_ext_len(routing(pk_original)) AND
    routing_type(routing(pk_transmitted)) =
      routing_type(routing(pk_original)))

```

```

destination1_l : LEMMA
  (pk_transmitted = transmission(pk_original) AND
   find_header(pk_original,destination1))
IMPLIES
  (find_header(pk_transmitted,destination1) AND
   next_header(destination1(pk_transmitted)) =
    next_header(destination1(pk_original)) AND
   hdr_ext_len(destination1(pk_transmitted)) =
    hdr_ext_len(destination1(pk_original)))

destination2_l : LEMMA
  (pk_transmitted = transmission(pk_original) AND
   find_header(pk_original,destination2))
IMPLIES
  (find_header(pk_transmitted,destination2) AND
   next_header(destination2(pk_transmitted)) =
    next_header(destination2(pk_original)) AND
   hdr_ext_len(destination2(pk_transmitted)) =
    hdr_ext_len(destination2(pk_original)) AND
   options(destination2(pk_transmitted)) =
    options(destination2(pk_original)))

payload_l : LEMMA
  pk_transmitted = transmission(pk_original)
IMPLIES
  payload(pk_transmitted) = payload(pk_original)

result : THEOREM
  ((pk_transmitted = transmission(pk_original)
   IMPLIES
    (version(IPv6(pk_transmitted)) =
     version(IPv6(pk_original)) AND
     priority(IPv6(pk_transmitted)) =
     priority(IPv6(pk_original)) AND
     flow_label(IPv6(pk_transmitted)) =
     flow_label(IPv6(pk_original)) AND
     payload_length(IPv6(pk_transmitted)) =
     payload_length(IPv6(pk_original)) AND
     next_header(IPv6(pk_transmitted)) =
     next_header(IPv6(pk_original)) AND
     source_address(IPv6(pk_transmitted)) =
     source_address(IPv6(pk_original))))
   AND
   ((pk_transmitted = transmission(pk_original) AND
    find_header(pk_original,hop_by_hop))

```

```

IMPLIES
  (find_header(pk_transmitted, hop_by_hop) AND
   next_header(hop_by_hop(pk_transmitted)) =
     next_header(hop_by_hop(pk_original)) AND
   hdr_ext_length(hop_by_hop(pk_transmitted)) =
     hdr_ext_length(hop_by_hop(pk_original)) AND
   jumbo_payload_length(hop_by_hop(pk_transmitted)) =
     jumbo_payload_length(hop_by_hop(pk_original))))
AND
  ((pk_transmitted = transmission(pk_original) AND
   find_header(pk_original, routing))
IMPLIES
  (find_header(pk_transmitted, routing) AND
   next_header(routing(pk_transmitted)) =
     next_header(routing(pk_original)) AND
   hdr_ext_len(routing(pk_transmitted)) =
     hdr_ext_len(routing(pk_original)) AND
   routing_type(routing(pk_transmitted)) =
     routing_type(routing(pk_original))))
AND
  ((pk_transmitted = transmission(pk_original) AND
   find_header(pk_original, destination1))
IMPLIES
  (find_header(pk_transmitted, destination1) AND
   next_header(destination1(pk_transmitted)) =
     next_header(destination1(pk_original)) AND
   hdr_ext_len(destination1(pk_transmitted)) =
     hdr_ext_len(destination1(pk_original))))
AND
  ((pk_transmitted = transmission(pk_original) AND
   find_header(pk_original, destination2))
IMPLIES
  (find_header(pk_transmitted, destination2) AND
   next_header(destination2(pk_transmitted)) =
     next_header(destination2(pk_original)) AND
   hdr_ext_len(destination2(pk_transmitted)) =
     hdr_ext_len(destination2(pk_original)) AND
   options(destination2(pk_transmitted)) =
     options(destination2(pk_original))))
AND
  ((pk_transmitted = transmission(pk_original))
IMPLIES
  payload(pk_transmitted) = payload(pk_original)))

END fragmentation

```

B The Proof Files

B.1 The Proof File of Type Specifications

```
(|frag_types|
  (|no_frag_packet_TCC1| "" (USE "no_frag_packet_as") NIL)
  (|frag_packet_TCC1| "" (USE "frag_packet_as") NIL))
```

B.2 The Proof File of Fragmentation and Reassembly

```
(|fragmentation|
  (|fragment_packets_assumption_TCC1| "" (SUBTYPE-TCC) NIL)
  (|fragment_packets_assumption_TCC2| "" (SUBTYPE-TCC) NIL)
  (|fragment_packets_assumption_TCC3| "" (GRIND) NIL)
  (|fragment_packets_assumption_TCC4| "" (GRIND) NIL)
  (|fragment_packets_assumption_TCC5| "" (SUBTYPE-TCC) NIL)
  (|fragment_packets_assumption_TCC6| "" (SUBTYPE-TCC) NIL)
  (|fragment_packets_assumption_TCC7| "" (GRIND) NIL)
  (|fragment_packets_assumption_TCC8| "" (GRIND) NIL)
  (|fragment_packets_assumption_TCC9| "" (SUBTYPE-TCC) NIL)
  (|rm_frag_assumption_TCC1| "" (SUBTYPE-TCC) NIL)
  (|rm_frag_assumption_TCC2| "" (SUBTYPE-TCC) NIL)
  (|get_header_assumption_TCC1| "" (SUBTYPE-TCC) NIL)
  (|hs_1| "" (USE "assembly_assumption")
    (("" (USE "get_header_asmp")
      (("" (USE "full_payload_asmp")
        (("" (USE "get_payloads_asmp")
          (("" (USE "order_fragments_asmp")
            (("" (SKOSIMP*) (("" (GRIND) NIL))))))))))))))
  (|header_set_1| "" (LEMMA "hs_1")
    (("" (SKOSIMP*)
      (("" (SPLIT)
        (("1" (GRIND) NIL) ("2" (GRIND) NIL) ("3" (GRIND) NIL)
        ("4" (GRIND) NIL))))))
  (|Ipv6_1| "" (USE "assembly_assumption")
    (("" (USE "get_header_asmp")
      (("" (USE "full_payload_asmp")
        (("" (USE "get_payloads_asmp")
          (("" (USE "order_fragments_asmp")
            (("" (SKOSIMP*)
              (("" (SPLIT)
                (("1" (GRIND) NIL) ("2" (GRIND) NIL)
                ("3" (GRIND) NIL) ("4" (GRIND) NIL))
```

```

        ("5" (GRIND) NIL) ("6" (GRIND) NIL))))))))))
(|hopbyhop_1| "" (USE "assembly_assumption")
  (" (USE "get_header_asmp")
    (" (USE "full_payload_asmp")
      (" (USE "get_payloads_asmp")
        (" (USE "order_fragments_asmp")
          (" (SKOSIMP*)
            (" (SPLIT)
              ("1" (LEMMA "header_set_1") (("1" (GRIND) NIL)))
              ("2" (GRIND) NIL) ("3" (GRIND) NIL)
              ("4" (GRIND) NIL))))))))))
(|routing_1| "" (USE "assembly_assumption")
  (" (USE "get_header_asmp")
    (" (USE "full_payload_asmp")
      (" (USE "get_payloads_asmp")
        (" (USE "order_fragments_asmp")
          (" (SKOSIMP*)
            (" (SPLIT)
              ("1" (LEMMA "header_set_1") (("1" (GRIND) NIL)))
              ("2" (GRIND) NIL) ("3" (GRIND) NIL)
              ("4" (GRIND) NIL))))))))))
(|destination1_1| "" (USE "assembly_assumption")
  (" (USE "get_header_asmp")
    (" (USE "full_payload_asmp")
      (" (USE "get_payloads_asmp")
        (" (USE "order_fragments_asmp")
          (" (SKOSIMP*)
            (" (SPLIT)
              ("1" (LEMMA "header_set_1") (("1" (GRIND) NIL)))
              ("2" (GRIND) NIL) ("3" (GRIND) NIL))))))))))
(|destination2_1| "" (USE "assembly_assumption")
  (" (USE "get_header_asmp")
    (" (USE "full_payload_asmp")
      (" (USE "get_payloads_asmp")
        (" (USE "order_fragments_asmp")
          (" (SKOSIMP*)
            (" (SPLIT)
              ("1" (LEMMA "header_set_1") (("1" (GRIND) NIL)))
              ("2" (GRIND) NIL) ("3" (GRIND) NIL)
              ("4" (GRIND) NIL))))))))))
(|payload_1| "" (USE "assembly_assumption")
  (" (USE "get_header_asmp")
    (" (USE "full_payload_asmp")
      (" (USE "get_payloads_asmp")
        (" (USE "order_fragments_asmp")

```

```

      (("" (SKOSIMP*) (("" (GRIND) NIL)))))))))
(|result| "" (SKOSIMP*)
((" (SPLIT)
  ("1" (LEMMA "Ipv6_1") (("" (INST?) NIL)))
  ("2" (LEMMA "hopbyhop_1") (("" (INST?) NIL)))
  ("3" (LEMMA "routing_1") (("" (INST?) NIL)))
  ("4" (LEMMA "destination1_1") (("" (INST?) NIL)))
  ("5" (LEMMA "destination2_1") (("" (INST?) NIL)))
  ("6" (LEMMA "payload_1") (("" (INST?) NIL))))))

```

VITA

İbrahim Şahin was born on December 14, 1970 in Ankara, Türkiye. He attended Gazi University in Ankara, Türkiye where he obtained the degree of Bachelor of Science in Electronics and Computer Education in June 1993. Right after graduating, he started working at a high school as a teacher. After teaching for six months at Isparta Technical and Vocational High School, he passed a nation wide competitive qualification examination organized by The Higher Educational Council of Turkey (YOK) and received a university sponsored grant toward the higher degree of M.S. and Ph.D. in an overseas country. After the examination he became a research assistant at Abant İzzet Baysal University in Bolu, Türkiye. In 1995, he moved to Norfolk, Virginia and started his M.S. study at Old Dominion University. He received his M.S degree from Old Dominion University in December 1997.

Currently, he plans to attend the Ph.D. program in the Department of Electrical and Computer Engineering at North Carolina State University.

His hobbies include traveling, hiking, skydiving and listening to new age music.