Testing Intra-Domain Routing in a Network Simulator

Diplomarbeit

von Dirk Jacob

Angefertigt am
Lehrstuhl für Computer Networking
Fachrichtung 6.2 - Informatik
Universität des Saarlandes, Saarbrücken

Prof. Dr. A. Feldmann

28. Januar 2002

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel verwendet habe.

Saarbrücken, 28. Januar 2002

Dirk Jacob

Kurzfassung

Um zuverlässige Aussagen über das Verhalten von großen Netzwerken bei Konfigurationsänderungen machen zu können, benötigt man Simulationen von solchen großen Szenarien. Insbesondere die Simulation des Routing und der Routing Protokolle spielen hier eine zentrale Rolle. Das Scalable Simulation Framework (SSF) ist ein Netzwerk Simulator, der ausdrücklich zur Simulation solch großer Netze entwickelt wurde. SSF beinhaltet auch Implementierungen der beiden weit verbreiteten Routing Protokolle BGP und OSPF. Wegen der zentralen Rolle des Routing muss sichergestellt werden, dass die Implementierungen dieser Protokolle konform zu deren Spezifikationen sind. Diese Arbeit gibt einen Überblick über die Konzepte des Konformitätstestens und stellt Methoden zur Testfallgenerierung vor. Routing Protokolle – speziell OSPF – werden im Hinblick auf ihre Testbarkeit untersucht und mehrere Test Suiten werden vorgestellt. Schließlich wird ein konzeptioneller Rahmen zum Testen in der SSFNet Simulationsumgebung vorgeschlagen. Innerhalb dieses Rahmens wird eine OSPF Test Suite implementiert.

Abstract

In order to make predictions about how large networks behave on configuration changes, one is dependent of having appropriate simulations of such large scenarios. Especially the correct simulation of routing and the routing protocols found in the Internet is crucial for these simulations. The Scalable Simulation Framework (SSF) is a network simulator, which was explicitly designed for the simulation of large networks and includes implementations of the two widely used routing protocols BGP and OSPF. Because routing plays such a central role, it must be assured, that the implementations of the routing protocols are conformant to their specifications. This thesis discusses different approaches to conformance testing and presents methods for the selection of test cases. Routing protocols in general and OSPF in special are analyzed with respect to conformance testing and several OSPF test suites are presented. Finally a conceptual framework for testing within SSFNet simulation framework is proposed and a test suite for OSPF is implemented using these concepts.

Contents

List of Figures

List of Tables

1. Introduction

"Oooh, they have the internet on computers now!"
– Homer J. Simpson

1.1. Motivation

The configuration and management of large IP networks, including the configuration of routing and setup of routing policies requires a deep understanding of all the effects which may arise, if the configuration changes or errors occur. There is also a strong interest in the global effects of such local changes in the topology or configuration. A network administrator may for instance wish to be able to predict the way over which packets are routed after changing the cost of a single link.

To understand all these effects even in big scenarios, scalable simulation tools are needed, which are suitable for small scenarios as well as for very large networks. A simulator, which was built to meet these requirements is the *scalable simulation framework* (SSF). It provides a generic scalable framework for the simulation of event-driven models. An additional layer of components models the functionality of networking components like hosts and routers as well as the internet protocols on top of the SSF framework. This framework of components is called *SSFNet*.

Strong attention must be payed to the correct simulation of routing, because only if routing is simulated correctly, the results of the simulation can be used to predict how a real network behaves. The Internet consists of a set of autonomous networks, which may interact with each other following certain policies. These autonomous networks are called *autonomous systems* (AS) and often represent the network of one special ISP. When we look at routing, we must distinguish *inter-domain* and *intra-domain* routing, that means the routing between the ASs and the routing internal to an AS.

In the Internet, routing protocols are used in order to be able to react dynamically on topology changes or failure situations. There are routing protocols, which are designed especially for inter-domain routing (so called *external gateway protocols* – EGP) and routing protools designed for the routing internal to an AS (*internal gateway protocols* – IGP). SSFnet supports both – EGP and IGP. BGP4 is an EGP which is widely used in the Internet and which is also implemented and well tested in SSFnet. SSFnet also includes a "static" version of OSPF, which is a popular IGP. This implementation omits all dynamic features, but can be used to import BGP routes into an AS. A full featured implementation of OSPF for the SSFnet environment is currently under devel-

opment at the Computer Networking Group at Saarland University. To assure, that the implementation properly realizes the OSPF standard, a conformance test suite must be provided as well. In the verification of a distributed protocol like OSPF there are different approaches and testing methods, which can be used.

An essential step in the design of a test suite is the selection of appropriate test cases. The decision must be made, whether to use a formal approach and prove formally if a given implementation is correct or rather use an intuitive approach which gives only a high confidence that the implementation works properly. The different approaches must now be analyzed with respect to testing of routing protocols in general and OSPF in special. For this, a general approach for testing routing protocols is proposed and applied to the OSPF protocol. For the testing of real network devices, several test devices are available. In addition, lab testing plays a central role in the testing process of many vendors. It must be evaluated, if the test suites used in practice are also applicable to the SSFNet environment. Therefore, general concepts for testing within SSFNet are discussed and a conceptual testing framework is proposed. Finally an OSPF test suite is implemented within this framework.

1.2. Structure of this work

Chapter 2 provides an overview of the scalable simulation framework and the SSF network models. After that, the different testing methods are discussed in Chapter 3. Chapter 4 gives an overview of the Open Shortest Path First routing protocol with a special respect to testing. Chapter 5 presents some test suites for OSPF testing and discusses how OSPF is tested in practice. One of these test suites was developed at the InterOperability Laboratory (IOL) at University of New Hampshire, where it is used by vendors of network products in the testing of their equipment. This test suite was implemented in the SSFnet environment to test OSPF. This implementation is described in Chapter 6 and the results are presented. Chapter 7 finally summarizes the results of this work.

2. SSF Network Models

In order to investigate, how local changes in a subnet affect the rest of the internet, appropriate simulations of very large networks with a large number of hosts, running different protocols, are needed. Simulations permit experimenting with configurations, which can hardly be done in an operative network. Moreover, simulations can help in the design of large networks, so reliable predictions about the behavior of a network can be made without having to build the network first.

Most of the current simulation tools can only deal with a small number of hosts and are often focused on limited problems [CNO99b]. When trying to simulate networks with many thousands of heterogeneous nodes, a special focus has to be scalability – scalability in terms of computing power as well as modeling scalability. The *scalable simulation framework* (SSF) together with the *SSF network models* (SSFNet) provides a simulation framework for such large scenarios. To ensure, that routing is simulated correctly, the implementations of routing protocols provided with SSFNet have to be tested. But to be able to test the routing protocols, the concepts of the simulation framework must be understood. This Chapter gives an introduction into the basic concepts used in SSFNet.

2.1. Overview

The SSFNet project covers two main areas:

- research on simulations and simulation software and
- network research.

The work on software research is focused on scalability, including modeling scalability as well as performance scalability. This work results in simulation tools, which are appropriate for network research. Network research includes research on the dynamic behaviour of very large networks, like research on phenomena which can only be seen in sufficiently large networks, predictions of the behaviour of the internet under alternative-future scenarios, etc. SSFNet emerged from the S3 project and is sponsored by the Defense Advanced Research Projects Agency (DARPA)[1], the Institute for Security Technology Studies at Dartmouth[2] and Renesys Corp[3]. The objective of the S3 project was,

[1] http://www.darpa.mil
[2] http://www.ists.dartmouth.edu
[3] http://www.renesys.com

to achieve improvements in scalability, speed and manageability of modeling and simulations of very large scale, multiprotocol networks with over 100,000 multiprotocol nodes.

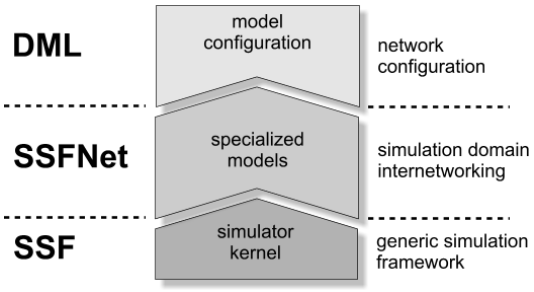

Figure 2.1.: SSFNet building blocks

Computing scalability can be achieved by using efficient parallel simulation techniques. These are kept strictly separated from the simulation domain and are implemented in a simulator kernel, which implements a standardized interface to access its functionality. The simulation domain is now modeled by providing appropriate components which are built on top of this interface. In simulation of networks for example, these are components which are modeling hosts, routers, links or the incorporated protocols. Such a component is called a *model*. Each model has a certain functionality and has to be configured appropriately. Obviously it is not a good idea to keep the configuration in the model itself. The model should rather implement only the functionality. The configuration of a specific simulation (e.g. the topology, configuration of hosts, etc.) has to be kept separately. This leads to the architecture, which is shown in Figure 2.1.

The Scalable Simulation Framework (SSF) is a parallel discrete-event simulator, which is not restricted to the simulation of communication networks, but is capable of simulating everything that can be modeled in terms of processes and event exchanges. SSF provides a compact interface for building simulations, which contains only five core classes (see Section 2.2). The application domain (e.g. the networking world) is modeled on top of this API, using an ordinary programming language like Java or C++. A set of models for the networking world (e.g. hosts, routers, links, protocols) is provided by the SSF network models (see Section 2.3). Each SSFnet model simulates the behaviour of a networking component or a protocol. When simulating a complex scenario, there may be thousands of instances of these models, which each have to be configured individually. For this, SSFnet models are provided with a possibility to configure the models at run-time, querying a configuration database. The model configuration is specified, using the *Domain Modeling Language* (see Section 2.4).

2.2. The Scalable Simulation Framework (SSF)

As seen above, SSF is not a dedicated network simulator – it can rather simulate everything that can be modeled in terms of processes and event exchange. The *SSF application programming interface* (SSF API), which consists of the five core classes `Entity`, `Event`, `inChannel`, `outChannel` and `process` hides all the internal simulator details and provides an easy-to-use interface to the simulator kernel. The application domain is implemented by a set of *models*, which are "standard C++ or Java programs, that derive new component classes that extend `Entity`, `Event` and `process` and use the framework to establish channel mappings and deliver events" [CNO99b].

2.2.1. Entity

The base class for all simulation components is the `Entity`. It serves as a container mechanism for the definition of alignment relations to other entities. When two simulation components work closely together, there are many events, which are exchanged between these components (e.g. the IP and TCP protocols on the same host). In a parallel simulation it would produce a communication overhead, if such tightly coupled components would be executed on different processors. Therefore, the `Entity` provides methods to indicate, that components should be aligned together. Examples for entities in the networking world are Hosts, Routers, Links, etc. The complete class interface is shown in Appendix A.1.

2.2.2. Event

`Event` is the base class for a single bit of information which is exchanged between entities. Protocol packets for instance are modeled as events. The complete interface of the `Event` class is shown in Appendix A.2.

2.2.3. inChannels and outChannels

Events can only be transmitted over channels, which are the endpoints of communication and belong to a specific entity. Every `outChannel` is mapped to one or more `inChannels`. Also several `outChannels` can be mapped to the same `inChannel`. This allows the realization of multicast input- and output channels. Additionally each `outChannel` can have associated a transmission delay, which determines the minimal time, an event needs to travel over the channel to its destination entity. This allows to model propagation delays on a link. The class interfaces of `in-` and `outChannels` is shown in Appendix A.3 and A.4.

2.2.4. Process

The fifth and last class is the `process`, which is used for modeling an entity's behaviour, e.g. a protocol's functionality. Each process is associated with one specific entity and has the possibility

1. of waiting on input to arrive on an `inChannel`

2. of waiting for some amount of time or

3. of simply waiting forever.

The class interface for the `process` class is shown in Appendix A.5.

2.2.5. How a simulation works

The simulation is started by calling one of the entities' `startAll` method. The framework then subsequently calls the other entities' `startAll` methods.

After the `startAll` method has been called, the first thing to happen is the initialization of all processes and entities. Therefor, the framework calls their `init` methods. The order of initialization may be implementation dependent and may even occur concurrently. After the processes are initialized, they are ready to start. The framework now executes the processes' `action` methods repeatedly until the end of the simulation is reached. When a process has to wait for a condition, it is suspended by the framework and resumed again, once the condition has been met.

2.2.6. Example

The following simple example should illustrate the concepts of SSF. Consider two hosts, each having a network interface which is able to send and receive single bits on a link. The hosts can be considered as entities, the network interfaces each consist of an `inChannel` and an `outChannel` and the bits, sent over a link are `Events`. The protocol running on the two hosts, which manages the exchange of bits can be modeled by a `process` (see Figure 2.2). The protocol realized in this example is a simple packet counter, which sends out packets periodically and counts incoming packets. The source code is shown in Figure 2.3.

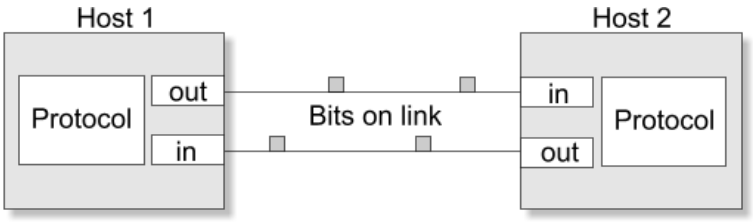

Figure 2.2.: SSF example

The `ExampleHost` class extends the SSF `Entity`. Each host has an attribute which counts the received events (line 3). The network interface consists of two channels – an `inChannel` called `IN` and an `outChannel` called `OUT` (lines 4 - 5). When an `ExampleHost` object is created, it first resets the event counter. Then the `inChannel` and the `outChannel` are constructed (lines 8 - 9). The `outChannel` has a delay associated with it. After that, two processes are created, which determine the host's runtime behaviour and implement the protocol. The first process simply waits for events on the `inChannel` (line 12) and

```
 1: import com.renesys.raceway.SSF.*;

 2: public class ExampleHost extends Entity {
 3:     public int rcvd;
 4:     public inChannel IN;
 5:     public outChannel OUT;

 6:     public ExampleHost() {

 7:         rcvd = 0;
 8:         IN = new inChannel( this );
 9:         OUT = new outChannel( this, 20 );

10:         new process(this) {    // receiver process
11:                 public void action() {
12:                     waitOn( IN );
13:                     rcvd++;
14:                 }
15:             };

16:         new process(this) {    // sender process
17:                 public void action() {
18:                     OUT.write(new Event());
19:                     waitFor( 20 );
20:                 }
21:             };
22:     }

23:     public static void main(String[] argv) {
24:         ExampleHost host1 = new ExampleHost();
25:         ExampleHost host2 = new ExampleHost();

26:         host1.OUT.mapto( host2.IN );
27:         host2.OUT.mapto( host1.IN );

28:         host1.startAll( 200 );
29:         host1.joinAll();

30:         System.out.println("total received events for host1 = "
                            + host1.rcvd);
31:         System.out.println("total received events for host2 = "
                            + host2.rcvd);
32:      }
33:  }
```

Figure 2.3.: Source code for the SSF example

increments the counter for each received event (line 13). The second process is responsible for sending packets. It simply sends an event on the `outChannel` (line 18) and waits for some time until it sends the next packet (line 19).

To execute the model, its `main` method must be called. The `ExampleHost` class is the instantiated twice (lines 24 - 25) and the `outChannels` of each of the hosts are mapped to the `inChannels` of the respective other host. So both are able to exchange events (lines 26 - 27). After that, the simulation is started by calling the `startAll` method of one of the hosts (line 28). The main process is then suspended by calling the `joinAll` method (line 29). After the simulation is over, the main process is resumed and the number of received events is given out (lines 30 - 31).

The execution of this simple example delivers the output shown in figure 2.4. The first five lines describe the version of SSF which has been used. The next

```
$ java ExampleHost
------------------------------------------------------------------------
|       Raceway SSF 1.0b08 (11 November 2000)
|        Proprietary Internal Development Release
|       (c)2000 Renesys Corporation
------------------------------------------------------------------------
------------------------------------------------------------------------
| 1 timelines, 5 barriers, 40 events, 110 ms, 0 Kevt/s
------------------------------------------------------------------------
total received events for host1 = 9
total received events for host2 = 9
```

Figure 2.4.: Output of the SSF example

three lines give some brief summary about internals of the simulation (like the runtime and other statistics). The last two lines finally are the output from the `main` method of our model.

2.2.7. SSF implementations

Currently, there are four independent implementations of SSF, which implement the SSF API.

Raceway is a high-performance commercial implementation of the Java SSF API and is maintained by Renesys Corp.

JSSF is the reference implementation of the Java SSF API from Cooperating Systems Corp. This implementation is no longer supported.

CSSF is the reference implementation of the C++ SSF API from Cooperating Systems Corp.

DaSSF is a high-performance implementation of the C++ SSF API maintained at Dartmouth College.

2.3. SSFNet

A complete set of models based on Java-SSF for the simulation of large networks is provided by the SSFNet package, which is open source software and is distributed under the GNU General Public License[4]. The SSFNet models provide components for the simulation of networks at IP level and above and include models for hosts, routers, links and a framework for modeling protocols. SSFNet is divided into four sub-packages:

SSF.OS contains a framework for protocol modeling and components related to the operation system.

SSF.Net contains models for the hardware-components, like hosts, routers, links, etc.

[4] http://www.fsf.org/copyleft/gpl.html

SSF.Util.Random contains additional classes for the generation of multiple independent random number streams.

SSF.Util.Streams contains support for an efficient multi-point network monitoring infrastructure to collect streaming data.

The packages `SSF.OS` and `SSF.Net` add an additional layer of abstraction and hide all details of the underlying SSF simulator API, allowing the implementation of protocols similar to a real operating system. A number of protocols, including IP, UDP, TCP, BGP4, OSPF and others, have already been implemented on top of this framework and are shipped with the SSFNet distribution (see Section 2.3.4).

To allow the separation of model functionality and configuration (see Section 2.1), all SSFNet models are *self-configuring*. That means, that each instance of a SSFNet model (e.g. every single host, router, etc. in a simulation scenario) can configure itself at runtime by querying a configuration database, which is specified using the Domain Modeling Language (see Section 2.4).

2.3.1. SSF.OS

The SSF.OS package contains a framework for modeling and simulation of network protocols. The architecture of this framework is based on the ideas of the x-kernel (see [OP91] and [HP92]), but is simpler. The framework is based on three classes – `ProtocolSession`, `ProtocolMessage` and `ProtocolGraph`.

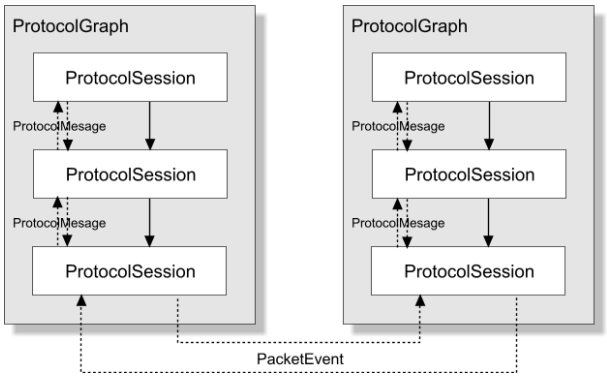

Figure 2.5.: The SSF.OS framework

The `ProtocolSession` is the base class for all classes, which provide protocol services. It defines basic mechanisms for requesting protocols services and for the handling of incoming mesages. Each `ProtocolSession` is a node in a `ProtocolGraph`, which is a directed acyclic graph and whose edges define a "*depends on*" relation (e.g. OSPF depends on IP). The `ProtocolSessions` communicate by exchanging `ProtocolMessages`. Each `ProtocolMessage` can have a payload associated with it, which must again be a `ProtocolMessage`. `ProtocolMessages` can travel in both directions along an edge between two

ProtocolSessions. When transmitted over a SSF channel, ProtocolMessages are transformed into SSF events, using the class PacketEvent.

Besides of these basic classes, the SSF.OS framework contains mechanisms for modeling timers (the Timer class) and blocking methods (the Continuaton interface).

ProtocolGraph

The ProtocolGraph serves as a container for all protocol services in a host or a router. When a simulation starts, it creates the ProtocolSessions and initializes them. The ProtocolGraph also provides the means to locate and access the individual ProtocolSessions given the protocol's name. See Appendix B.2 for the complete class interface of the ProtocolGraph.

ProtocolSession

Each ProtocolSession is a node in a ProtocolGraph and contains mechanisms for initialization, configuration and access to the services of the protocol implemented by this ProtocolSession. Besides of real protocols, ProtocolSessions may be used to model different applications, which are running in a host, such as different clients or servers (like HTTP-servers or HTTP clients). In addition, lower level services, like network interfaces, which are used by ProtocolSessions can also be modeled as ProtocolSessions.

The handling of incoming protocol messages is controlled by the push method. This method can be used by other ProtocolSessions to push ProtocolMessages into this ProtocolSession. The ProtocolSession can now determine the sending session by evaluating the fromSession parameter of the push method and process the message appropriately. The only thing, a push method must never do is blocking. To avoid blocking, Timers and Continuations (see below) can be used. The complete interface of the ProtocolSession class is described in Appendix B.1.

ProtcolMessage

The ProtocolMessage class is the base class for modeling protocol packets. A packet consists of a doubly linked list of ProtocolMessages (see Figure 2.6). SSF.OS reduces memory and processing overhead, using zero-copy processing of ProtocolMessages. That means that the ProtocolMessages are not serialized, when transmitted over the simulated network. Only references of the objects are transmitted. This means that ProtocolMessages must be handled carefully, when they are processed (e.g. when retransmitting packets, which may have been changed on their way, a copy constructor has to be used). The interface of the ProtocolMessage class is described in Appendix B.3.

PacketEvent

ProtocolMessages are not SSF events and so can not be transmitted over SSF channels. When a ProtocolMessage has to be transmitted from one

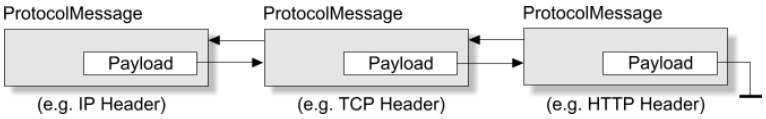

Figure 2.6.: Structure of a protocol packet

`ProtocolGraph` to another the `PacketEvent` class must be used, which contains a `ProtocolMessage` as payload. The interface of the `PacketEvent` class is described in Appendix B.4.

Timers and Continuations

To avoid deadlocks and other undesired behaviour, a `ProtocolSession`'s code is not allowed to call the `SSF.process` methods `waitFor` and `waitOn`. But when processing packets, a `ProtocolSession` must be able to wait for certain events. For the implementation of such a behaviour, SSF.OS provides the two classes `Timer` and `Continuation`. The `Timer` class serves as a means to model single shot timers. The action must be defined in a callback method for each timer. Appendix B.5 shows the complete interface of the `Timer` class. In order to model blocking methods, the `Continuation` interface can be used, which is shown in Appendix B.6.

2.3.2. SSF.Net

The SSFNet models for modeling and simulation of network elements, like hosts, routers, network interfaces and links are contained in the `SSF.Net` package. The main classes included in this package are `Net`, `Host`, `Router`, `NIC` and `link`.

Net

The `Net` class is a container for all other components of the simulation. It is used for loading the whole network's configuration and controls the initialization of the whole simulation. The `Net` class also controls the assignment of IP addresses and allows a recursive composition of large networks.

Host

The `Host` class is derived from `ProtocolGraph`, which is described in Section 2.3.1. Each `Host` must by definition contain at least the IP protocol and one network interface.

Router

The `Router` class is derived from the `Host` class, but provides no additional features. At the moment[5] it serves mainly as a means to distinguish between ordinary hosts and routers.

[5]The current SSFnet release is version 1.4.0alpha.

NIC

The `NIC`[6] class is derived from the `ProtocolSession` (see Section 2.3.1) and provides a pseudo-protocol to implement network interfaces. Each `NIC` maintains a pair of `in-` and `outChannels` to communicate with the world outside the `ProtocolGraph`.

Link

The `link` class defines a layer 2 link between 2 or more hosts. Depending on the number of hosts an appropriate link layer is used for event exchange. The `ptpLinkLayer` (point-to-point link layer) class is used for links with only two attached interfaces, the `lanLinkLayer` is used for bus-style links with more than 2 attached `NIC`s. Each `linkLayer` provides a pair of in- and outChannels for each attached interface. When a `PacketEvent` arrives on a link, the appropriate `outChannel` for this packet is chosen and the packet is sent out of it.

Additional classes

`SSF.Net` provides additional helper classes and interfaces, which are used inside the classes described above. These additional classes and interfaces include support for packet queues, routing tables, etc.

2.3.3. SSF.Util

The `SSF.Util` package consists of two sub-packages:

`SSF.Util.Random` contains classes for generation of random number streams and different probability distributions. For random numbers, the high quality random number generators from the package `cern.jet.random` are used. `SSF.Util.Streams` provides support for recording of multiple data streams for monitoring purposes.

2.3.4. Included protocols

SSFNet also provides models for some important protocols, which are based on the SSF.OS framework (see Section 2.3.1). Detailed documentation for all the protocols is provided with the SSFNet source code[7].

IP

The `IP` class implements a subset of the IP protocol. It provides support for routing and forwarding of IP packets (`IPHeaders`). Support for IP broadcast and multicast is not yet implemented.

[6] NIC = Network Interface Card.
[7] See the SSFNet website http://www.ssfnet.org for details.

UDP

The User Datagram Protocol (UDP) [Pos80] is implemented in the package `SSF.OS.UDP`. Like the TCP implementation, this package also contains simple servers and clients which can be used for the simulation of streaming UDP client-server sessions. These are contained in the sub-package `SSF.OS.UDP`.

TCP

The package `SSF.OS.TCP` contains a complete implementation of the Transmission Control Protocol (TCP) [Pos81]. There is also an extensive test suite, which is used for validation testing of the implementation. Besides these tests, the package `SSF.OS.TCP.test` contains models for simple TCP servers and clients for simulation of simple file transfers.

Socket

This package includes classes and interfaces, which model a BSD socket interface for the implementation of various applications. There are two implementations of these socket interface – TCP- and UDP-sockets. The `udpSockets` are contained in the UDP package, the `tcpSockets` are contained in the TCP package.

HTTP

The `SSF.OS.WWW` package contains classes for modeling of traffic generated by HTTP clients and servers. This package is still under development.

BGP

The package `SSF.OS.BGP4` implements the Border Gateway Protocol (BGP) as specified in RFC 1771 [RL95]. BGP is the de facto standard for inter-domain routing in the internet. The BGP implementation comes with a complete test suite, which validates the required behaviours.

"Static" OSPF

The package `SSF.OS.OSPF` contains a partial implementation of the Open Shortest Path First (OSPF) intra-domain routing protocol. Wide parts of the OSPF standard deal with dyamic behavior like neighbor discovery. These dynamic features are not supported in this version of OSPF, since all information about the network topology can be obtained from the network's configuration database. The main task for this implementation of OSPF is the flooding of routes obtained from an external routing protocol such as BGP.

OSPFv2

A complete implementation of OSPF which supports the whole OSPF standard [Moy98] is currently under development at the Computer Networking Group at

Saarland University. A closer description of this package can be found in Section 6.2.

2.4. Domain Modeling Language (DML)

The Domain Modeling Language (DML) is a description language, which is used by SSFNet for automatic model-configuration. DML serves as a means to describe the topology of the simulated network as well as the configuration of the different components. DML provides mechanisms for inheritance and attribute substitution. A collection of networking components or even networks with common configurations can be kept in a *dictionary*. Large networks can now be composed, using components from this dictionary. This concept makes DML flexible and powerful enough to model even large networks.

2.4.1. Syntax

Each DML expression consists of a list of whitespace separated key/value pairs. Keys are alphanumerical strings – reserved keys begin with an underscore character. A value can either be a string or another DML expression, which is enclosed in square brackets. Value strings are arbitrary ASCII strings enclosed in double quotes. These quotes may be omitted, if the string contains neither whitespaces nor square brackets. Value strings may be regular expressions –any special characters must be escaped with a backslash character. The grammar of DML is shown in figure 2.7.

```
DML          ::=  (space* Attribute space*)+
Attribute    ::=  Key space+ Value
Key          ::=  Name | ReservedName
Name         ::=  ^[^_][a-zA-Z0-9_-]*
ReservedName ::=  ^_[a-zA-Z0-9_-]*
Value        ::=  String | \[ DML \]
String       ::=  [^(space)\]\[]+ | "[^"]+"
space        ::=  [ \t\n]
```

Figure 2.7.: DML grammar

Currently, there are three reserved names in DML: `_schema`, `_find` and `_extends`. Figure 2.8 shows a DML configuration for a router with 4 interfaces, running the IP and OSPF protocols.

2.4.2. Schemas

The syntax of DML makes no assumptions on the application domain, which is modeled. The syntax allows arbitrary nested structures and arbitrary names as well as arbitrary value strings. But each application domain will have certain rules which have to be followed, concerning the nesting structure, attribute names and value ranges. Especially the design of very large scenarios requires a possibility to check a given DML configuration automatically for its correctness in the application domain.

```
_schema .schemas.router
router [
    id 1
    interface [id 0 bitrate 100000000 latency 0.0001]
    interface [id 1 bitrate 100000000 latency 0.0001]
    interface [id 2 bitrate 100000000 latency 0.0001]
    interface [id 3 bitrate 100000000 latency 0.0001]
    graph[
        ProtocolSession[name ip use SSF.OS.IP]
        ProtocolSession[name ospf use SSF.OS.OSPF.OSPF]
    ]
]
```

Figure 2.8.: DML example

```
valuetype ::=  %format
format    ::=  [SFI] count |
               S count: regexp |
               T count: keypath
count     ::=  empty | integer(!)? |
               integer<integer | <integer
integer   ::=  [1-9][0-9]*
```

Figure 2.9.: Schema format

For this, DML provides a mechanism called *schema checking*. A *schema* is a DML attribute tree, which defines attribute names, value types and the allowed number of attribute instances. The syntax for a schema is defined in Figure 2.9. The schema checker validates an attribute's syntactic correctness, when the `_schema` attribute is specified. This attribute has a *keypath* argument, which specifies where to find the schema. A keypath is a dot-separated list of keys, which represents an attribute in a DML configuration.

```
schemas [
  router [
    graph %T1!:.schemas.graph

    interface [
        id %I1!
        ip %S1!
        bitrate %I1!
        latency %F1!
    ]
  ]
  graph [
     ...
  ]
]
```

Figure 2.10.: DML schema example

Figure 2.10 shows an example of a schema for the router shown in Figure 2.8. The `graph` attribute specifies, that each router has exactly one graph, which is specified in the dictionary under `.schemas.graph`. The next lines specify, that each router also has at least one interface attribute with the given subattributes. Each router interface has exactly one `id` attribute, whose value is a interger number. There is also exactly one `ip` attribute, which contains a

string value. The `bitrate` and `latency` attributes each are contained exactly once in each interface – the `bitrate` has an integer value, `latency` has a float value.

2.4.3. Attribute substitution

DML supports the substitution of attributes with the keyword `_find` (*has-a* relation). Like the `_schema` keyword, `_find` has a keypath argument, which specifies the attribute to insert in the DML specification. The last attribute in that keypath is imported in the DML configuration.

```
dictionary [
  routers_common [
    graph [
      ...
    ]
  ]
]

Net [
  router[
    id 0
    _find .dictionary.routers_common.graph
]
```

When the `_find` in the example above is expanded, a the result is the following DML configuration:

```
Net [
  router[
    id 0
    graph [
      ...
    ]
]
```

When multiple instances of an attribute are present (e.g. multiple graph attributes in the `routers_common` section above), `_find` returns an enumeration which contains all those instances.

2.4.4. Inheritance

Another important feature of DML is inheritance (*is-a* relation), which is realized with the `_extends` attribute. This attribute also takes a *keypath* parameter, which specifies, which attributes to inherit. The following example should illustrate the DML inheritance semantics:

```
dictionary [
  pc [
    cpu pentiumIV
    mem 256MB
    pointing_device mouse
    application word
    application webbrowser
  ]
  cad [
    pointing_device tablet
    application acad
  ]
```

```
]

computer [
  _extends .dictionary.pc
  _extends .dictionary.cad
]
```

After evaluation, the computer attribute is equivalent to:

```
computer [
  cpu pentium IV
  mem 256MB
  pointing_device mouse
  application word
  application webbrowser
  pointing_device tablet
  application acad
]
```

DML allows multiple inheritance, that means inheriting attributes from multiple keys. The ordering of attributes stays intact, when multiple instances of an attribute are inherited. That means, that the attributes appear as if they had been defined directly at the position of the `_extends` statement.

2.5. Execution of a simple simulation

The following simple example should illustrate the execution of a SSFNet simulation. Look at the scenario given in Figure 2.11.

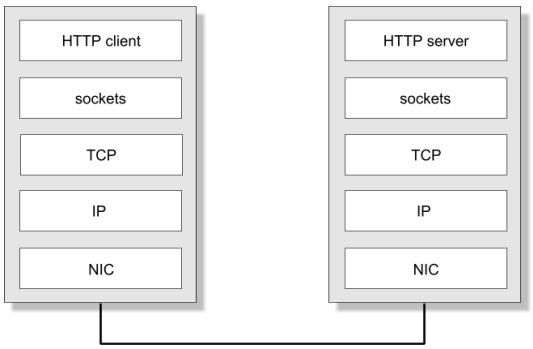

Figure 2.11.: Example network

The network consists of only two hosts, each of them having one network interface. One of the hosts is a HTTP server, which is running IP, TCP, sockets and a HTTP server. The other host, which is a HTTP client is also running IP, TCP and sockets. Additionally it runs an HTTP client. The hosts have a link between them.

Let us look at the DML configuraton of this example:

```
_schema [_find .schemas.Net]
Net [
    frequency 1000000000

    host [
```

```
            id 1
            interface [
                id 0
                _extends .dictionary.host_config.interface100Mbit
            ]
            graph [
                ProtocolSession [name client use SSF.OS.WWW.httpClient
                    _extends .dictionary.client_config
                ]
                _extends .dictionary.sockets_graph
            ]
        ]
        host [
            id 2
            interface [
                id 0
                _extends .dictionary.host_config.interface100Mbit
            ]
            graph [
                ProtocolSession [name client use SSF.OS.WWW.httpServer
                    _extends .dictionary.server_config
                ]
                _extends .dictionary.sockets_graph
            ]
        ]
        link [attach 1(0) attach 2(0) delay 0.01]

        traffic [ pattern [ client 1 servers [nhi 2(0) port 80] ] ]
    ]
```

As we saw in Chapter 2.3.2, the whole simulation is initialized and controlled by the `SSF.Net.Net` class. So the topmost attribute of the DML configuration must be a `Net` attribute. It contains a sub-attribute called `frequency` which specifies the time resolution of the simulation (1ns here). Next, there are two `host` attributes, which contain the configuration of the two hosts. Each of the hosts has an `id` which must be unique within the enclosing `Net`. Each of the two hosts also has an interface, whose properties are inherited from the 100Mbit interface specified in the dictionary. Additionally, each host must have a protocol graph. The common part of this protocol graph is defined in the `sockets_graph` attribute in the dictionary and is inherited by both of the hosts. Besides that, the first host conains the `httpClient` protocol in its graph, the second host contains the `httpServer` protocol. After the host configuration is complete, a link between the two hosts is configured. Then, a `traffic` attribute specifies, that the client communicates with the web-server. Let us now have a closer look on the dictionary, which contains the configuration details:

```
    dictionary [
        sockets_graph [
            ProtocolSession [name socket use SSF.OS.Socket.socketMaster]
            ProtocolSession [name tcp use SSF.OS.TCP.tcpSessionMaster]
            ProtocolSession [name ip use SSF.OS.IP]
        ]

        client_config [
            start_time 1.0
            inter_session_time [
                distribution [name "Exponential" lambda 0.01]
            ]
            pages_in_session [
                distribution [name "Exponential" lambda 0.2]
            ]
            inter_page_time [
```

```
            distribution [name "Pareto" k 25.0 alpha 2.0]
        ]
        inter_request_time [
            distribution [name "Pareto" k 0.1667 alpha 1.5]
        ]
        _find .dictionary.http_options.http_hdr_size
        persistent false
        show_report true
    ]

    server_config [
        port 80
        client_limit 10
        objects_in_page [
            distribution [name "Pareto" k 0.6667 alpha 1.2]
        ]
        object_size [
            distribution [name "Pareto" k 2000.0 alpha 1.2]
        ]
        response_delay [
            distribution [name "Exponential" lambda 10.0]
        ]
        _find .dictionary.http_options.http_hdr_size
        show_report true
    ]
    http_options [
        http_hdr_size 1000
    ]
    host_config [
        interface100Mbit [bitrate 100000000 latency 0.0001]
    ]
]
```

The `socket_graph` contains three `ProtocolSessions` – IP, TCP and the sockets API – which are inherited by both of the hosts' protocol graphs. The `client_config` and `server_config` attributes contain specific configuration details for the `httpClient` and `httpServer` protocol's configuration. These include appropriate configuration of the randomly distributed response times, and other parameters of the HTTP sessions. Common HTTP options are found in the `http_options` attribute, from where they are included via attribute substitution. Finally, the `host_config` attribute contains the interface configuration parameters for the 100 MBit interface that is used in both of the hosts.

2.6. Simulating a large network

The previous section showed at a fine level, how a simulation is configured. This section finally shows, how large simulations can be built using DML for the description of the topology and configuration of the components. The network we want to simulate is shown in figure 2.12. It is composed of 10 smaller networks, which are interconnected by a backbone. Each of this smaller networks again consists of 10 routers. Six of those routers each have two LANs connected to them – one LAN with 2 webservers and one LAN with 20 client hosts. That makes a sum of 1420 routers and hosts. The complete network is a single autonomous system, using OSPF for the distribution of routing information.

Large networks are usually composed of smaller networks and components which are kept in a directory. Since larger networks are composed of smaller ones, the LANs for the servers and the client hosts are modeled first. The

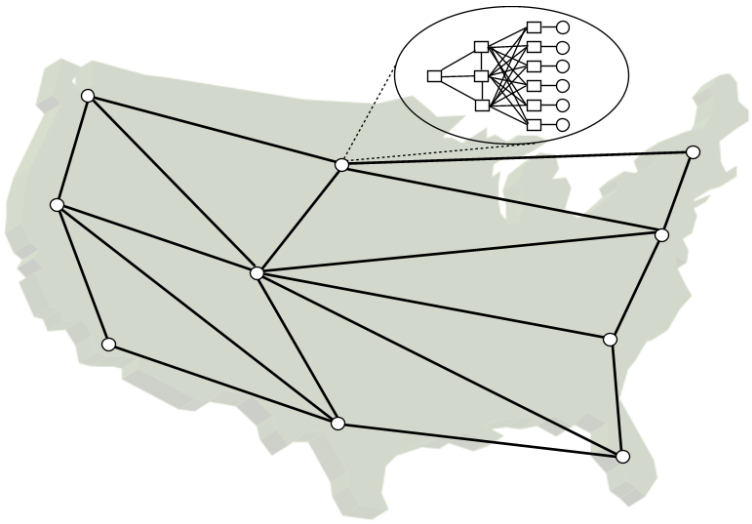

Figure 2.12.: Simulation of a large network

basic configurations of clients and servers are also kept in the directory. Since they were shown in the previous Section, they are omitted here for the sake of simplicity. Common router configurations are kept in the directory, too. These define routers with a different number of interfaces.

```
dictionary [
    networks [
        ...

        servers_clients [
            router[ id 1 _extends .dictionary.routers.rt_5 ]

            # the servers
            host[ idrange [ from 2 to 3 ]
                  nhi_route [dest default interface 0 next_hop 1(3)]
                  _extends .dictionary.hosts.server
            ]

            # the client hosts
            host[ idrange [ from 4 to 23 ]
                  nhi_route [dest default interface 0 next_hop 1(4)]
                  _extends .dictionary.hosts.client
            ]

            # the servers lan
            link[ attach 2(0) attach 3(0) attach 1(3) delay 0.0005 ]

            # the hosts lan
            link[ attach 4(0) attach 5(0) attach 6(0) attach 7(0)
                  attach 8(0) attach 9(0) attach 10(0) attach 11(0)
                  attach 12(0) attach 13(0) attach 14(0) attach 15(0)
                  attach 16(0) attach 17(0) attach 18(0) attach 19(0)
                  attach 20(0) attach 21(0) attach 22(0) attach 23(0)
                  attach 1(4) delay 0.0005
            ]
        ]
    ]
]
```

The LAN's are connected to a router, which has five interfaces. Three of them

are later connected to the rest of the network. Interfaces 4 and 5 are connected to the servers' and the clients' LANs. The OSPF configuration of this router is not specified explicitly, so it will look for a default area configuration in one of the enclosing networks. The clients and servers all run the same configuration and can all be configured at once, using the `idrange` attribute. Once all hosts, servers and routers are defined, they must be interconnected, using the `link` atribute.

The next step in building the large network is now to interconnect the LAN's and prepare the links to the backbone. Since the OSPF configuration and the number of interfaces to the backbone differ in the 10 networks, the DML configurations are also different. But there is a common part, which is the same in all networks. This concerns the configuration of links between all routers and between the routers and subsidiary networks. It is important to notice, that this part of the dictionary cannot be used outside the networks' configurations, because it relies on the existence of routers that are configured appropriately.

```
dictionary [
    networks [
        ...

        net_common [
            Net[
                idrange [ from 1 to 6 ]
                _extends .dictionary.networks.servers_clients
            ]

            # links to the border router
            link[ attach 2(6) attach 1(0) delay 0.0001 ]
            link[ attach 3(6) attach 1(1) delay 0.0001 ]
            link[ attach 4(6) attach 1(2) delay 0.0001 ]

            link[ attach 2(7) attach 3(7) delay 0.0001 ]
            link[ attach 4(7) attach 3(8) delay 0.0001 ]

            # links to the networks
            link[ attach 2(0) attach 1:1(0) delay 0.0001 ]
            link[ attach 2(1) attach 2:1(0) delay 0.0001 ]
            link[ attach 2(2) attach 3:1(0) delay 0.0001 ]
            link[ attach 2(3) attach 4:1(0) delay 0.0001 ]
            link[ attach 2(4) attach 5:1(0) delay 0.0001 ]
            link[ attach 2(5) attach 6:1(0) delay 0.0001 ]

            link[ attach 3(0) attach 1:1(1) delay 0.0001 ]
            link[ attach 3(1) attach 2:1(1) delay 0.0001 ]
            link[ attach 3(2) attach 3:1(1) delay 0.0001 ]
            link[ attach 3(3) attach 4:1(1) delay 0.0001 ]
            link[ attach 3(4) attach 5:1(1) delay 0.0001 ]
            link[ attach 3(5) attach 6:1(1) delay 0.0001 ]

            link[ attach 4(0) attach 1:1(2) delay 0.0001 ]
            link[ attach 4(1) attach 2:1(2) delay 0.0001 ]
            link[ attach 4(2) attach 3:1(2) delay 0.0001 ]
            link[ attach 4(3) attach 4:1(2) delay 0.0001 ]
            link[ attach 4(4) attach 5:1(2) delay 0.0001 ]
            link[ attach 4(5) attach 6:1(2) delay 0.0001 ]
        ]
    ]
]
```

First, all the LANs are defined, using the configuration discussed above. Then all links are defined to interconnect routers contained in this network.

This common section does not yet define any routers. This is done in the next part of the DML configuraton. Since the router configuration differs for all the networks, there has to be one such configuration for each of the networks.

```
dictionary [
    networks [
        net1 [
            default_area[
                id 1
                _extends .dictionary.protocols.ospf_defaults
            ]

            router[ id 1 _extends .dictionary.routers.rt_3if_3bb
                graph[
                    _extends .dictionary.protocols.router_graph
                    ProtocolSession [
                        _extends .dictionary.protocols.ospf_net1_b
                    ]
                ]
            ]
            router[ id 2 _extends .dictionary.routers.rt_8if
                graph[
                    _extends .dictionary.protocols.router_graph
                    ProtocolSession [
                        _extends .dictionary.protocols.ospf_net1_8if
                    ]
                ]
            ]
            router[ id 3 _extends .dictionary.routers.rt_9if
                graph[
                    _extends .dictionary.protocols.router_graph
                    ProtocolSession [
                        _extends .dictionary.protocols.ospf_net1_9if
                    ]
                ]
            ]
            router[ id 4 _extends .dictionary.routers.rt_8if
                graph[
                    _extends .dictionary.protocols.router_graph
                    ProtocolSession [
                        _extends .dictionary.protocols.ospf_net1_8if
                    ]
                ]
            ]
            _extends .dictionary.networks.area_common
        ]
    ]

    protocols [
        ...

        ospf_defaults [
            if [
                id 0
                network ptp
                cost 16
            ]
            ...
        ]
        ospf_net1_b [
            name OSPF use SSF.OS.OSPFv2.OSPF

            area [
                id 0
                stub false

                if [
                    id 0
                    network ptp
```

```
                    cost 128
                ]
                ...
            ]
            area [
                id 1
                stub false

                if [
                    id 4
                    network ptp
                    cost 16
                ]
            ]
        ]
    ]
]
```

First, the OSPF default configuration included and an area ID is assigned. After that, the router is configured, which connects the network to the backbone. From the set of common router configurations, one with three backbone links is chosen and its OSPF session is configured appropriately. After that, the remaining routers are configured, again using common router configurations from the dictionary (i.e. with 8 and 9 interfaces respectively). Networks 2 to 10 are configured similar to this network. After the networks are defined, they can be connected by the backbone links taken from the dictionary (`.dictionary.networks.area_common`).

```
Net [
    frequency 1000000000

    Net[ id 1 _extends .dictionary.networks.net1 ]
    Net[ id 2 _extends .dictionary.networks.net2 ]
    Net[ id 3 _extends .dictionary.networks.net3 ]
    Net[ id 4 _extends .dictionary.networks.net4 ]
    Net[ id 5 _extends .dictionary.networks.net5 ]
    Net[ id 6 _extends .dictionary.networks.net6 ]
    Net[ id 7 _extends .dictionary.networks.net7 ]
    Net[ id 8 _extends .dictionary.networks.net8 ]
    Net[ id 9 _extends .dictionary.networks.net9 ]
    Net[ id 10 _extends .dictionary.networks.net10 ]

    # the backbone links
    link[ attach 1:1(3) attach 2:1(3) delay 0.0001 ]
    link[ attach 1:1(4) attach 4:1(3) delay 0.0001 ]
    link[ attach 1:1(5) attach 5:1(3) delay 0.0001 ]
    link[ attach 2:1(4) attach 3:1(3) delay 0.0001 ]
    link[ attach 2:1(5) attach 5:1(4) delay 0.0001 ]
    link[ attach 2:1(6) attach 6:1(3) delay 0.0001 ]
    link[ attach 3:1(4) attach 6:1(4) delay 0.0001 ]
    link[ attach 4:1(4) attach 5:1(5) delay 0.0001 ]
    link[ attach 4:1(5) attach 7:1(3) delay 0.0001 ]
    link[ attach 4:1(6) attach 8:1(3) delay 0.0001 ]
    link[ attach 5:1(6) attach 6:1(5) delay 0.0001 ]
    link[ attach 5:1(7) attach 8:1(4) delay 0.0001 ]
    link[ attach 5:1(8) attach 9:1(3) delay 0.0001 ]
    link[ attach 5:1(9) attach 10:1(3) delay 0.0001 ]
    link[ attach 6:1(6) attach 10:1(4) delay 0.0001 ]
    link[ attach 7:1(4) attach 8:1(5) delay 0.0001 ]
    link[ attach 8:1(6) attach 9:1(4) delay 0.0001 ]
    link[ attach 9:1(5) attach 10:1(5) delay 0.0001 ]
]
```

The top-level network can now be modeled, using the networks defined above. First of all, the time resolution for this simulation has to be configured. It is set

to 1ns. Then the 10 networks are defined and interconnected by appropriate links.

2.7. Limitations

SSFnet provides a scalable framework for modeling and simulation of large networks and comes with a variety of protocols. Anyhow, there are some limitations, which are briefly dicussed in this section. SSFnet does not yet provide models for the simulation of link layer protocols (e.g. Ethernet). Additionally IP broadcast and multicast are not implemented yet. Another limitation of SSFnet is its static configuration. The topology and network components are configured at the beginning of the simulation. This configuration is maintained until the simulation ends. Dynamic behavior, such as link failures, routers and hosts being shut down or restarted can not be simulated easily.

Although these featues are useful in certain contexts, the main scope of SSFnet is the simulation of large networks and the analysis of routing and traffic flow. Current research investigates, how different traffic flows influence each other. Dynamic changes in routing are not yet considered, so these features do not play a central role. The link layer, IP broadcast and multicast primarily have local effects, so they can also be diregarded. However, most of the features are planned to be implemented in future versions of SSFnet.

2.8. The Role of Testing in SSFnet

For a simulation it is crucial to imitate the behavior of the reality as precise as possible. Hence it is important, that all components behave properly and that protocol standards are implemented correctly. To verify their correctness, a set of tests is provided with all the protocols implemented in SSFnet, which show if an implementation behaves correctly. Since new versions of SSFnet are released periodically another important application of these tests is regression testing. This assures that changes in the framework itself do not have any negative effects on the protocol implementations.

3. Conformance Testing

To guarantee that different implementations of a protocol are able to interoperate, they must strictly fulfill the requirements given by the protocol specification. Additionally, when simulating a protocol, it must be assured, that the simulation behaves exactly as the real world. Again, this is achieved by strictly following the protocol specification. *Conformance Testing* is the process of verifying, if a given implementation performs exactly as required in the specification. The complete set of tests which must be performed to make a statement about the conformance of a protocol is called a *test suite*.

For most of the implementations it is impossible to observe the internal states of the implementation. The first problem is, that most of the protocols do not require an interface for the observation of internal states. The second problem is that different implementations may use additional states, so that each implementation would require an own test suite. For this reason, the whole implementation is regarded a black box. Tests are performed by validating if the implementation produces an accurate output for every possible input.

A logical approach to testing would be to feed the *implementation under test* (IUT) with every possible input combination in order to observe if it produces the right output. This approach is called *exhaustive testing*, which is very hardly to perform with complex protocols, because the number of test cases would be too many. Instead of testing all possible combinations of input, only the important ones are taken into account, e.g. the input combinations which drive a protocol implementation from one state to another. The design of a test suite for a given protocol has then to go through the following procedure:

1. identify fault models and important input combinations
2. identify test purposes according to the fault models and the protocol specification
3. design test cases for each of the test purposes
4. draw conclusions of the tests - so exhaustive testing can be avoided

3.1. Categories of Tests

According to the extent to which an indication about the conformance of a protocol implementation to its specification is provided, there are several categories of testing which must be distinguished.

3.1.1. Basic Interconnection Tests

This category of tests is used to verify that an IUT supports some basic features of a standard. If these tests are passed, there is a minimum degree of conformance which allows the implementation to interconnect with another implementation of the same standard.

The basic interconnection tests are adequate for the detection of severe offences against the protocol's specification and as a first set of tests to perform before more detailed and more expensive tests are used. These tests are not able to provide a conformance statement for their own, therefore more detailed tests are necessary.

3.1.2. Capability Tests

These tests are used to assure that certain minimum capabilities, the so called *static conformance requirements* are provided by the IUT. These minimum capabilities permit two implementations to inter-operate with each other. They can be used to subdivide the protocol into several functional units, as well as to define the range of parameters or timers, which must be supported. Capability tests do not include tests for the optional features of a protocol.

3.1.3. Behavior Tests

While the static conformance requirements define the minimum capabilities, which are mandatory for an implementation, the *dynamic conformance requirements* describe the behavior which is permitted by the specification. That means, that they include all the optional behaviors, an implementation may perform. Behavior tests are designed in order to test these dynamic requirements and are – together with the capability tests – a basis for the conformance statement.

3.1.4. Conformance Resolution Tests

This group of tests finally provides definite statements about whether an implementation fulfills certain requirements or not. They provide a means to check if particular features have been implemented correctly or not and may be based on capability and behaviour tests.

3.1.5. Other Test Types

In addition to the four test types described before, which are used in order to tell if an implementation conforms to its specification, there are various other types of tests, which may be performed with an IUT. *Performance tests* for instance are used to measure how good (or how bad) an implementation performs under various conditions. *Stress tests* may be used to verity that an implementation still performs at all, when exposed to extreme situations (e.g. heavy load, lots of incoming packets). *Robustness tests* may make a statement about how well the IUT recovers from error conditions. *Regression tests* are used, to determine

if an implementation still works as expected after changes (e.g. bug fixes, new features added, etc.).

3.2. Open Systems Interconnection – Testing Methodology

In 1983, the International Organization for Standardization (ISO) began on standardizing test methods for protocols, which were developed in the context of the Open System Interconnection (OSI) – Basic Reference Model [Int94]. This work resulted in the international standard ISO IS 9646 "OSI Conformance Testing Methodology and Framework" [Int83].

The standard consists of seven parts. Part 1 covers the general concepts of conformance testing, while part 2 deals with test suite specification and test system architectures. Part 3 defines a test notation and part 4 covers the test realization. The last three parts finally deal with the means of testing and organizational aspects. The basic concepts given in the ISO 9646 standard are introduced in this section.

3.2.1. General Concepts

In the OSI Conformance Testing Methodology and Framework (CTMF), all implementations to be tested are implementations of OSI protocol entities. In reality, the concepts are also applicable to other systems. Nevertheless, the following text uses the terminology from the OSI world to avoid ambiguities.

The OSI reference model (OSI-RM) provides a framework for the specification of communication services and protocols [LU94a]. The reference model consists of seven hierarchical layers each providing an additional level of abstraction to the layers above. An entity (i.e. a protocol or a service) at layer N provides (N)-*services* to the layer above. Two entities at the same layer, that communicate with each other are called *peer entities*. An entity, providing (N)-services is called a (N)-*service provider*. Peer entities at layer N exchange (N)-*protocol data units* ((N)-PDUs). The rules and conventions for this exchange are called an (N)-*protocol*. The logical interface between (N)- and $(N+1)$-entities which exposes the (N)-service to the $(N+1)$-entity is called the (N)-service access point ((N)-SAP). At the SAPs the two entities interact by exchanging (N)-abstract service primitives ((N)-ASPs). See figure 3.1 for an illustration of the terms defined above.

Each implementation of a protocol entity has two interfaces, through which it interacts with its environment – the upper and the lower interface. OSI conformance testing considers the *implementation under test* (IUT) as a black box and performs testing only through interactions at these interfaces. The interfaces are called *Points of Control and Observation* (PCOs). PCOs are usually found at the (N)- and $(N-1)$-service access points, when testing a N-layer protocol.

The OSI conformance testing methodology and framework distinguishes between an *upper tester* (UT) and a *lower tester* (LT). The lower tester usually

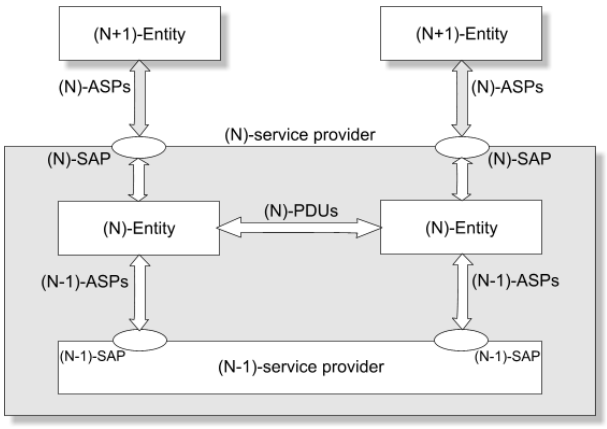

Figure 3.1.: (N)-entities and (N)-service providers

controls and observes the abstract service primitives which are exchanged with the lower interface of the IUT. The upper tester exchanges ASPs with the upper interface. The actions of the upper- and lower testers are coordinated in a suitable way.

3.2.2. Test Procedure

The procedure for conformance testing is outlined in the CTMF standard. Static and dynamic conformance requirements are taken into account, as well as the *protocol implementation conformance statement* (PICS). The PICS is a document, which describes the capabilites and options actually implemented in the IUT and serves as a basis for the selection of test cases. Additional information about the IUT is given in the *protocol implementation extra information for testing* (PIXIT). This document contains extra information which is required for testing of the IUT, such as information about the hardware running the IUT, adresses, phone numbers and other information which may be required to run the tests.

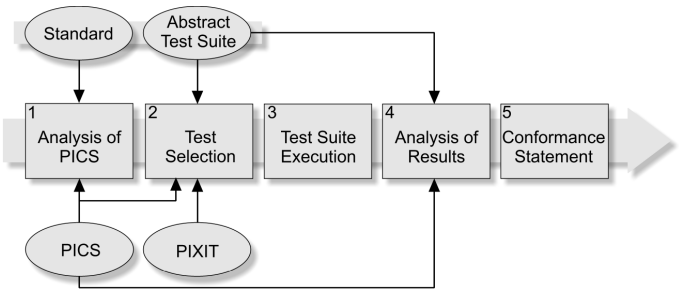

Figure 3.2.: ISO conformance testing procedure

There are five steps to perform in the conformance testing procedure. First, the PICS must be analysed with regard to its own consistency and to consistency

with the static conformance requirements for the protocol.

As a second step, the appropriate test cases must be selected from the conformance test suite (more information on abstract test suites is given in section 3.2.3), using the information from PICS and PIXIT. The test cases are then parametrized, using the information from the PIXIT. The abstract test cases must be converted into executable test cases, which can be executed on the real testers. The result of this step is an *parametrized executable test suite*.

In the third step, the executable test suite is actually executed. This is done in three phases, which have already explained before: basic interconnection tests, capability tests and behaviour tests.

After the execution of the test suite, the results must be analyzed. Therefore the output of the test cases is validated with respect to the according test purposes and the valid behaviour of an implementation. In the analysis step, verdicts must be made, whether a test case has been passed, failed or whether the result is inconclusive.

In the last step, the results of the tests are subsumed with the results from the analysis of the PICS in order to validate that the test results are also consistent with the PICS. Finally a conclusion about the conformance of the IUT to the requirements of the protocol specification must be made. The results of the tests, as well as the conformance statement are recorded in a standardized test report.

3.2.3. Abstract Test Suite Specification

In the context of CTMF, *abstract test suites* are used for conformance testing. These test suites define *abstract test cases*, which are independent of a particular IUT and describe the behaviour of arbitrary protocol implementations. This approach helps to design implementation independent test suites, which can also be standardized with the protocol specification, so all implementations can be tested under the same circumstances and run through the same tests. Each abstract test case has three components:

- the test *preamble*, which brings the IUT in the starting state for the test purpose,
- the test *body*, which defines the test events needed for this test purpose and
- the test *postamble*, which is used to bring the IUT into a stable, defined state after the test has been performed.

Dependent of the locations of the PCOs and the coordination of the tests, different abstract test methods can be distinguished. The location of a PCO is strongly dependent of the IUT.

The Local Method

In the *local method*, the PCOs are located at the upper and lower service boundaries of the IUT. Assumed, that the IUT is a (N)-entity, the test events are

(N)-ASPs at the upper tester and $(N-1)$-ASPs containing (N)-PDUs at the lower tester. Upper tester and lower tester are coordinated by suitable test coordination procedures. The local method can be used, when the LT, UT and the IUT are running on the same system. Because the LT and the IUT are exchanging (N)-PDUs, the LT must at least partially emulate the behaviour of an (N)-entity.

The Distributed Method

When the lower interface of an IUT is not accessible directly, the *distributed method* can be used. This method locates the upper tester at the upper service boundary of the IUT and the lower tester at the opposite site of the according $(N-1)$-service provider. Like in the local method, the UT also deals with (N)-ASPs as test events. The lower tester deals with the counterpart of the $(N-1)$-APSs on the remote side which are commonly called $(N-1)$-ASP"s. Again, the UT and LT are coordinated by some test coordination procedures. The LT and the IUT again communicate in terms of (N)-PDUs.

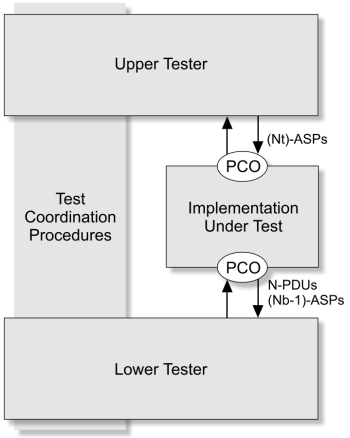

Figure 3.3.: The local method

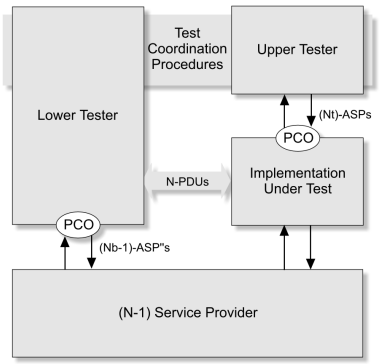

Figure 3.4.: The distributed method

The Coordinated Method

An enhanced version of the distributed method is given with the *coordinated method*. A standard UT is used and the UT and LT are coordinated by means of a *test management protocol* (TMP). The only PCO needed is located at the opposite side of the $(N-1)$-service provider and deals with test events specified in terms of $(N-1)$-ASP"s and (N)-PDUs. The lower tester also handles test management PDUs (TM-PDUs), which can be transferred in-band (that means, carried as data of the protocol to be tested) or out-of-band (carried by a lower-level protocol).

The Remote Method

The *remote method* does not need an upper tester at all. The lower tester, which is located at the remote side of the $(N-1)$-service provider, handles $(N-1)$-ASP"s and (N)-PDUs. Test coordination is managed manually by test operators, if tests have to be coordinated at all. The LT and the IUT are syncronized only by the means of (N)-PDUs.

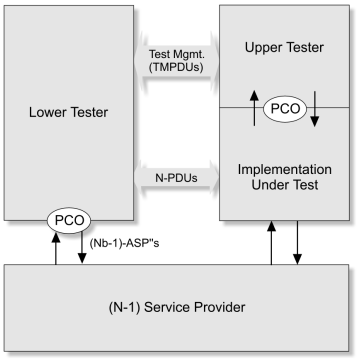

Figure 3.5.: The coordinated method

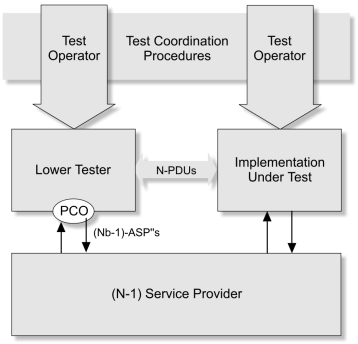

Figure 3.6.: The remote method

Other Methods

For the testing of more complicated implementations, such as the testing of all the layers of a multi-layers IUT as a whole, the above test methods are not sufficient. More sophisticated test methods which can deal with these IUTs are proposed in [Lin94] and [Ray94].

3.2.4. Tree and Tabular Combined Notation

The *tree and tabular combined notation* (TTCN) is a notation which was standardized in part 3 of the CTMF standard and is used as a notation for the specification of abstract test suites [SG00]. Test cases are described as event trees in which external behaviour of a protocol is described (e.g. when sending the message "connection request", either a "connection confirm" or a "disconnect indication" message will be received).

There are two different forms of TTCN – the graphical form (TTCN/gr), which is intended to be read by humans and the machine processable form (TTCN/mp), which was designed for the exchange of TTCN test suites and further automatic processing. Each form can be transformed into an equivalent representation of the other form, so only the human readable TTCN/gr is used in this section to illustrate the concepts of TTCN.

Each TTCN test suite consists of four parts, which are described below.

Overview Part

The *overview part* serves as a kind of a table of contents for the test suite. It provides information about the test suite structure, defines the test suite name and describes the test architecture which is used. It also provides additional references, like the protocol standard, the PICS and PIXIT documents.

Declarations Part

In the *declarations part* provides declarations and definitions which are used in the following parts of the test suite. This includes declaration of data types, variables, constants, timers, operations, selection expressions, test components, PCOs and the encoding of ASPs and PDUs. Data types can be defined, using the TTCN type system or alternatively using the ASN.1 data types. ASN.1 (abstract syntax notation one) is a standardized language for the specification of data types and is specified in the ISO standard 8822.

Constraints Part

The *constraints part* describes the values of PDUs and ASPs which can be sent to or may be received from the IUT. Each constraint descibes a PDU type defined in the declarations part and concrete values or value ranges for the PDU fields.

Dynamic Part

The dynamic part finally describes the dynamic behavior of a tester through the definition of test cases, test steps and default behavior descriptions. It describes the actual execution of tests and the assignment of verdicts. The test behavior is specified as a tree, which is represented using increasing levels of indentation to indicate progression of the test exexution. A sample behavior tree is found in Figure 3.7.

```
LT!SYN_Packet                    ; LT sends a SYN_Packet
    START wait_timer             ; start a timer
        LT?ACK_Packet            ; did the LT receive an ACK_Packet?
            LT!FIN_Packet        ; if yes, send a FIN_Packet
        ?TIMEOUT wait_timer      ; if the timer fires before receiving
                                 ; an ACK
            LT!FIN_Packet        ; send a FIN_Packet
```

Figure 3.7.: Sample TTCN behavior

3.3. Test Case Selection

The selection of appropriate test cases plays a central role in conformance testing. This step is critical for the fault coverage of a test suite. Because of the importance of this step, there are efforts to use formal techniques for test case

selection. Formal techniques have the advantage, that the correctness and the coverage of the resulting test suites can be mathematically proven.

3.3.1. Relating Protocols to Finite State Machines

Most of the formal techniques for test case selection presented in this chapter, require that the protocol is specified as a finite state machine (FSM). A FSM is a *4-tuple*

$$FSM = (S, I, O, T)$$

where

- S is a finite set of states,
- I is a set of inputs,
- O is a set of outputs and
- T is a set of transitions from state s_j to s_k:

$$T = \bigcup_j t_j : (s_j, i_j) \longrightarrow (s_k, o_j)$$

where $s_j, s_k \in S; i_j \in I; o_j \in O$.

That means that an FSM which is in state s_j and which is feeded with input i_j, produces an output o_j and goes over into state s_k. A commonly used representation for FSMs is the representation as a directed graph

$$G = (V, E)$$

where the set V contains one vertex v_j for each state $s_j \in S$. Each transition $t_j : (s_j, i_j) \longrightarrow (s_k, o_j)$ corresponds to an edge $e_{j,k} \in E$, which is has a label of i_j/o_j.

A graph is called *strongly connected* if for any two vertices v_j and v_k there is a *nonnull* sequence of edges from v_j to v_k. Following from the tight relation of FSMs to graphs, a FSM is called strongly connected, when there is an input sequence $i_j, i_{j+1}, \ldots, i_k$ which brings the FSM from state s_j to state s_k for each pair of states $s_j, s_k \in S$.

Testing a protocol which is specified by means of an FSM can now be reduced to testing the correct operation of each of the possible transitions $t_j \in T$. This leads to three basic test steps, which have to be performed for each transition $t_j : (s_j, i_j) \longrightarrow (s_k, o_j)$:

Step 1 bring the FSM into state s_j

Step 2 apply input i_j and observe, if the FSM correctly produces output o_j

Step 3 verify, that the IUT is in state s_k after the transition

3.3.2. Transition Tours

The first and most straightforward approach for the generation of test cases is the *transition tour* method [SvB94]. A transition tour is a sequence of inputs, which causes the FSM to exercise every transition of the FSM at least once, starting from the initial state. A *tour* in a graph is defined as a *nonnull* sequence of consecutive edges, that starts and ends at the same vertex. From this it follows, that a transition tour must also start and end at the initial state of the FSM. During the transition from one state to another, the output which is produced by the FSM is observed.

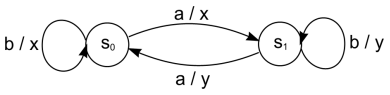

Figure 3.8.: Example FSM

As an example consider the FSM given in figure 3.8. A transition tour for this FSM must start at state s_0 and also end there. In between it has to go through every transition at least once. The input sequence b, a, b, a is a transition tour for this FSM.

There are methods which find transition tours for FSMs in a heuristic way, but the resulting tours are not optimal (i.e. not the shortest possible tours). The problem of finding minimum length tours in a graph is known as the *Chinese Postman Problem* and can be solved, when the according graph is strongly connected. As we saw earlier, transition tours can be represented as tours in the graph representation of the according FSM, which als has to be strongly connected for the solution of the problem. Thus the solution of the chinese postman problem for the graph representation of an FSM also is an optimal transition tour of the FSM. The solution of the chinese postman problem is described in [UD94].

The output of an FSM does not necessarily give a hint about the state of the FSM after a transition. This is a major problem of the transition tour approach because the tester has no means to verify, that the IUT is in the desired state after a transition tour. So the transition tour method cannot detect any errors due to transitions resulting in the wrong state, but it detects all errors in the output function.

3.3.3. Distinguishing Sequences

The problem of the transition tours method can be solved using the *distinguishing sequences* method. A distinguishing sequence is an input sequence of a FSM, which generates an output sequence, that uniquely identifies the state, the FSM was in before the input sequence was applied. Consider again the example FSM given in Figure 3.8. This FSM has a distinguishing sequence $DS = a, b$. The output produced by this input sequence uniquely identifies the state, the FSM was before the input sequence was applied.

State	output sequence
s_0	x, y
s_1	y, x

Table 3.1.: Output sequences generated by the DS

The concept of distinguishing sequences was originally introduced for testing of digital circuits. Some additional concepts are necessary in the context of distinguishing sequences, which are described in [Hen94]. A *synchronizing sequence* is an input sequence of an FSM whose application guarantees, that the FSM is in a certain state after the application, regardless of the state, it was in before. A *transfer sequence* for an arbitrary state $s_j \in S$ is the input sequence which brings the FSM from the initial state to state s_j. Regarding this, the basic test steps can be modified as follows:

Step 1 Bring the implementation in state s_j by applying the synchronizing sequence for the initial state of the FSM and after that applying the transfer sequence for state s_j.

Step 2 Apply the input i_j and observe, if the proper output o_j is generated.

Step 3 Now apply the distinguishing sequence and verify if the FSM has really been in state s_k by observing the generated output.

An algorithmic solution for finding a distinguishing sequence for a FSM is presented in [Gön94]. The major disadvantage of the presented algorithm is, that the FSM has to be *fully specified* to apply this method. So in every state s_j of the FSM, there must be a transition $t_n : (s_j, i_n) \longrightarrow (s_k, o_n)$ for each possible input $i_n \in I$. Real protocols are typically not fully specified. Another drawback is, that not every FSM must have a distinguishing sequence at all.

3.3.4. Characterizing Sequences (W-Method)

For FSMs which do not possess a distinguishing sequence, but are fully specified, the *characterizing sequences* method was introduced. This method was introduced for FSMs that are fully specified, but do not have a distinguishing sequence. A *characterizing set* for a FSM is a set of input sequences, such that the set of output sequences generated by the application of each of the input sequences in state s_j of the FSM uniquely identifies this state s_j. Each one of the input sequences in this characterizng set works partly like a distinguishing sequence by distinguishing s_j from a subset of other states. The application of all those input sequences so guarantees, that s_j can be distinguished from any other state. The characterizing set for a FSM is also called the *W-set* for this FSM – this method is therefor also called the *W-method*.

The FSM in Figure 3.9 does not have a distinguishing sequence [LU94b]. Consider an input sequence $W_1 = a, b, a$. Table 3.2 shows the output generated when applying W_1 to the FSM in its different states.

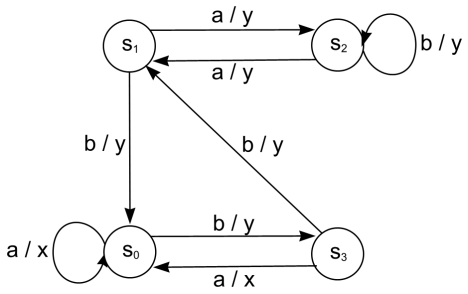

Figure 3.9.: Example FSM that has no distinguishing sequence

State	output sequence
s_0	x, y, x
s_1	y, y, y
s_2	y, y, x
s_3	x, y, x

Table 3.2.: Output sequences generated by W_1

So, applying W_1 can uniquely identify, if the FSM has been in state s_1 or in state s_2 – it cannot distinguish between the states s_0 and s_3. Therefore, consider the input sequence $W_2 = b, a$, which produces the output given in Table 3.3. As we can see, this input sequence can distinguish between the states s_0 and s_3. Now, we have input sequences by which we can determine all the states of the FSM. Therefore, the characterizing set of the FSM is $W = \{W_1, W_2\}$. The following steps must now be repeated for each input sequence W_j from the $W - set$:

Step 1 Apply a synchronizing sequence for the initial state and after that apply the transfer sequence for state s_j.

Step 2 Apply input i_j and observe, if the according otput o_j is produced.

Step 3 Apply W_j and verify, if the correct output sequence is generated.

A method for the generation of the $W - set$ is given in [Cho94]. From the definition of the characterizing sequence, we can see that every distinguishing

State	output sequence
s_0	y, x
s_1	y, x
s_2	y, y
s_3	y, y

Table 3.3.: Output sequences generated by W_2

sequence is also a characterizing sequence. Given a distinguishing sequence DS, we can immediately give the according characterizing set $W = \{DS\}$.

Partial W-Method

Test suites produced by the characterizing sequences method are usuall very long. There is another version of the W-method, which produces shorter test suites. It is called the *partial W-method* (or *Wp-method*) [FvBK+94]. Here, inputs which do not produce different outputs for different states are excluded from the W-set, because they are not giving any useful informaiton.

3.3.5. Unique Input/Output Sequences

A *unique input/output sequence* (UIO-sequence) for a state s_j is an input sequence whose output uniquely identifies s_j. A FSM with n states has at least n UIO-sequences, if it is strongly connected and minimal.

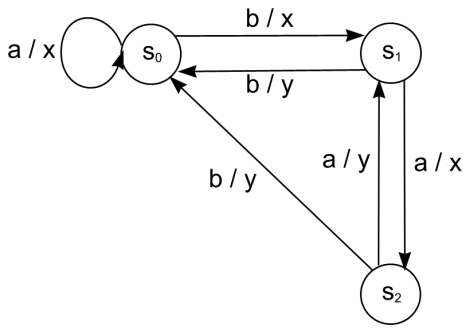

Figure 3.10.: Example FSM for UIO sequences

State	output sequence
s_0	b/x
s_1	$a, b/x, y$
s_2	a/y

Table 3.4.: UIO sequences

Consider the FSM specified in Figure 3.10. Each of the states of this FSM has a UIO sequence as showed in table 3.4. The modified test procedure for a transition using UIO sequences works as follows:

Step 1 Bring the FSM into state s_j.

Step 2 Apply input i_j and observe that output o_j is produced.

Step 3 Now apply the UIO sequence for state s_k, in which the FSM should be and observe the output.

But compared with the methods described earlier, UIO sequences follow a different approach. Distinguishing sequences and characterizing sequences can be applied to a FSM without regarding the state, it should be in. If a transition which has to be tested fails, these methods can at least tell in which state the FSM is now. UIO sequences on the other hand must be explicitly applied according to the state the FSM should be in. If the test has failed and the FSM is not in the desired state, UIO sequences can give no further information. The UIO method does not require that the FSM is fully specified and results in much shorter test sequences. Algorithms for the generation of UIO sequences are presented in [SD94] and [ADLU94].

3.4. Formal Description Techniques

As we saw before, nearly all methods for test case selection assume that the protocol is specified as a finite state machine or an extended finite state machine, which has certain properties (fully specified, strongly connected). This has the advantage, that there are no ambiguities in the specification, which can arise especially when the protocol is specified in natural language.

But if we look at real protocols, there is only a part of the protocol which is specified by a FSM. This part is normally the *control flow portion* of a protocol (e.g. establishing connections). The second part of a protocol specification is the *data portion* which deals with the data transmitted between two entities, such as parameter values, etc. This part is often not as precise as the control flow part, and cannot be modeled by the means of FSMs. Usually, this part is secified in natural language. In order to minimize ambiguities, there are efforts to use *formal description techniques* (FDT) to specify protocols, including the data part. The most important FDTs are described in this section.

3.4.1. Specification and Description Language

The Specification and Description Language (SDL) is an object-oriented, formal language, which is intended for the specification of complex, event-driven, real-time and interactive applications involving many concurrent activities that communicate using discrete signals. SDL can model both, the behavior and the data in a system. SDL is standardized by the International Telecommunications Union–Telecommunications Standardization Sector (ITU-T) as recommendation Z.100 [Int99a].

3.4.2. Message Sequence Charts

Message Squence Charts (MSC) are a graphical language, which is used for the specifcation of system traces. It is standardized by the ITU-T in recommendation Z.120 [Int99b]. MSCs are used for the specification of system requirements and are often used in combination with SDL.

3.4.3. Language Of Temporal Ordering Specification

The Language Of Temporal Ordering Specification (LOTOS) is another formal description technique which was developed to specify protocols. LOTOS is divided into two main parts [DCB91]. The first part models a system as a set of processes, that communicate with each other. The second part covers the description of data structures. LOTOS was standardized by the ISO in 1988 and is defined in the international standard ISO 8807.

3.4.4. Estelle

Estelle was published in 1989 as another formal description technique in the ISO standard ISO 9074. In Estelle, a protocol's behaviour is modeled as a set of extended finite state machines, which communicate with each other [FUA+99]. The Estelle language has a formal mathematical implementation-independent semantics to avoid ambiguities. Estelle specifications consist of two parts. The first part describes the architecture of the system, specifying a collection of subsidiary systems and modules. The second part finally describes the behavior of each of these modules, using extended finite state machines.

3.4.5. Test Case Selection based on FDTs

Specifications using Formal Description Techniques, such as SDL, LOTOS and Estelle, can be represented as *Labelled Transition Systems* (LTS) [GHNS93]. An LTS is an automaton, where all states can be interpreted as end states [GHN93]. In particular, an LTS is a tuple

$$LTS = (Q, E, R, q_0)$$

where

- S is a countable set of states,
- L is a countable set of events (labels),
- $T \subseteq S \times L \times S$ is a set of transitions and
- $s_0 \in S$ is the initial state.

This fact can be used for the selection of test cases for a protocol specification, that is given in one of these languages. Dependent of the specification language and the particular protocol specification, there are different methods for the derivation of test cases. Some of these methods are discussed in [GNSH93], [GL94] and [SvBC94].

3.5. Testing Distributed Systems

The testing of distributed systems has special requirements, which have to be taken into account. The complete process of testing is affected. The abstract

test architectures, presented in Section 3.2.3 must be extended as depicted in Figure 3.11 [GKSH98]. The *Implementation Under Test* (IUT) has several interfaces, over which it can communicate with its environment. Each interface corresponds to a *Point of Control and Observation* (PCO). To test the IUT, there is one *Parallel Test Component* (PTC) for each interface. A *Main Test Component* (MTC) is used to create the PTCs and to compute the final verdict. The PTCs coordinate themselves through the exchange of messages at *Coordination Points* (CP).

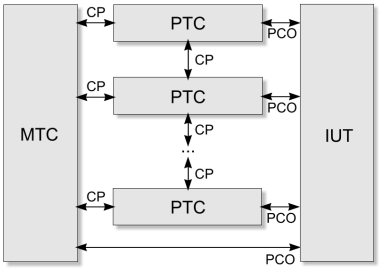

Figure 3.11.: Test architecture for distributed systems

Test cases are specified in TTCN in the OSI conformance testing framework. TTCN assumes, that each test case is executed sequentially. But distributed systems normally have concurrent behaviors, so the concepts of TTCN must be extended as well. These extensions are provided by *Concurrent TTCN* (C-TTCN) [BWC98], which provides explicit mechanisms for synchronization and communication between the test components.

3.6. Informal Testing

The methods and techniques described above allow a formal definition of test cases for a protocol specification. This leads to test suites whose fault coverage and correctness can be proven formally. But as seen before, these methods all have very strict requirements. Many methods for test case selection all require the protocol to be specified as a FSM. But when we look at real protocols, we see that only a part of them is specified as a FSM. Another drawback is, that the FSMs have certain restrictions, like the need to be fully specified. Real protocols often do not fit these requirements.

Another problem of using formal methods is that the protocol specifications are often written in natural language. Formal methods are used to avoid ambiguities. But when using formal techniques, the specifications have often to be translated into a formal description first, using one of the FDTs. So the ambiguities arise in this translation process rather than in testing.

The alternative to using formal methods is an informal approach. Test cases are selected by carefully reading the specification and selecting the appropriate test cases intuitively. Although the fault coverage and correctness of test suites designed like this cannot be proven, this method is widely used. The resulting test suites can – if designed carefully – give a high degree of certainty, that an

implementation which passes the tests will interoperate with other implementations of the protocol. The purposes of the test cases are intuitively clear and the test suites can be used early in the development process.

3.7. Comparison of the Methods

This Chapter showed a methodology for conformance testing of protocols. The coverage of a test suite strongly depends on the selection of test cases. Different formal methods for test case selection have been presented. But not each of these methods is applicable to all protocol specifications. In addition, the resulting test suites may have a different number of test cases. This section compares the different methods with respect to applicability, coverage and test suite length.

All formal methods make assumptions about the protocol specification. The methods related to FSMs for example require, that

- the FSM has to be fully specified,
- the FSM has to be strongly connected,
- the FSM must have a known and finite number of states and
- the set of inputs has to be known and finite.

Most protocols cannot be expressed as FSMs at all. FDTs must be used for the specification of such protocols. But many protocol specifications (e.g. most of the Internet protocol standards) have no standardized formal description. So, for the derivation of test cases with a formal method, the standard has to be translated first. During this process, ambiguities or misunderstandings may occur. The coverage of test suites, generated with formal methods can be proven. In fact, not all specifications can be covered completely [LDvB$^+$93], but at least he degree of coverage can be determined. The test suite length again strongly depends on the method for test case selection. When using informal methods, the coverage of the resulting cannot be proven. But informal methods can be used with every specification, regardless if it has a formal specification or not. This makes these methods attractive for the use with Internet standards.

4. Open Shortest Path First

The Internet consists of a collection of autonomous networks which are operated by the internet service providers (ISPs). Those autonomous networks are called *autonomous systems* (AS). If we look at the routing of IP packets in the Internet, we can distinguish between the routing internal to an AS and the routing between the ASs. Because of its huge size and the dynamics of the Internet it is not a good idea to use statically configured routes. Therefore, *routing protocols* are used to distribute routes over the network and to update them dynamically, when the topology changes.

Hence, in the simulation of large networks, it is crucial, that the routing protocols are simulated correctly, so that the data packets in the simulation are routed the same way as they are in a real network. To guarantee this, the routing protocols must comply to their specificaion. This can be assured by running appropriate tests which verify the protocols' behavior. As we saw in Section 2.3.4, the SSFNet simulation framework already contains a well tested implementation of the inter-domain routing protocol BGP4. But intra-domain routing is also of interest in a network simulation. For this, SSFnet is provided with an implementaion of a popular intra-domain routing protocol, called Open Shortest Path First (OSPF). This Chapter discusses the OSPF protocol with respect to testing an implementation for correctness.

4.1. Routing Protocols

Routing protocols are used to make routing decisions at run-time to be able to react on topology changes (e.g. link failures) in a timely fashion. The routing internal to an AS is controlled by *internal gateway protocols* (IGP), the routing between the ASs is controlled by *exterior gateway protocols* (EGP). This classification is done because IGP and EGP meet different requirements. An IGP is required to calculate efficient routes and to recalculate these routes quickly, when the network changes. EGPs, which are used to establish routing between different ISPs, are required to be able to express policies. That means, that some routes may be preferred over others although they may have a higher cost.

Another way to classify routing protocols is to distinguish the algorithms, which are used for the distribution of routes and the calculation of routing tables. There are two main categories for routing protocols: *Distance Vector Protocols* and *Link State Protocols*.

4.1.1. Distance Vector Protocols

Each router in a *distance vector routing protocol* knows its neighbors, the links to these neighbors and the cost (distance) to reach these neighbors. Each router now announces to its neighbors which destinations it can reach at which cost. When a router now finds out, that one of its neighbors has a better route to a particular destination, it updates its least-cost route to that destination and again announces this route to all neighbors. So each router knows the networks it can reach at which cost, but has no detailed information about the way the packets travel. It does neither know how much hops the packet takes nor does it know something about the other routers on its way.

4.1.2. Link State Protocols

Link state routing protocols are working differently from distance vector protocols. Routers generate packets with information about their own identity and all the routers and networks, they are connected to, including the cost to reach each of these routers or networks. These packets are then sent to all neighbors in the network. If some changes in the topology occur, such as links coming up or going down, the router generates a new packet and advertises its links again. These packets are distributed all over the network by a distributed algorithm. This procedure is called *flooding*.

The flooding procedure assures, that all routers on the network know the complete topology at any given time. So every router can calculate the shortest path to every destination, given this topology map, using some appropriate algorithm. When links fail, the topology change is discovered quickly and routing keeps intact. A property of link state algorithms is, that each router is able to compute the routing table of each other router in the network.

4.1.3. How to test a Routing Protocol

To assure, that a routing protocol is implemented correctly, it must be tested to behave exactly as defined in the protocol specification. Since routing protocols are complex systems, it is a good idea to divide the functionality into several independant functional areas. So the complexity of testing the whole protocol can be broken down into several simpler testing problems. This Section proposes such a general structure for routing protocols. According to this structure, several general test purposes can be formulated for arbitrary routing protocols.

Looking at existing routing protocols like RIP, OSPF or BGP [Doy98], some common functionality can be observed. All these prtocols have some notion of *neighborhood*. A router must assure, that it can communicate with its neighbors in order to send routing information and forward traffic. When a neighbor fails or the link between a router and one of its neighbors goes down, the router must notice this. Another common functionality in routing protocols is, that routers send routing information to their neighbors under certain circumstances (e.g. upon detection of link failures or discovery of better routes). Additionally a router must react appropriately when it receives such updates (e.g. the flooding procedure in link state potocols). The third common part of routing

protocols is the proper handling of the routing information, they receive. Each router must decide, whether the received information affects its own routing table. If necessary, the routing table must be recalculated, using the appropriate algorithm.

The testing procedure can take advantage out of this structure. The different functional areas can be tested mostly separate from each other. This reduces the complexity of the testing procedure significantly. As seen in Chapter 3, a prerequisite for testing is the identification of appropriate fault models and test purposes. At a high level, these can be formulated for an arbitrary routing protocol:

1. Does the router establish communication with its neighbors properly?
2. What happens if communication cannot be established?
3. What happens if the neighbor becomes unreachable due to a failure?
4. Does a router send the correct routing information to its neighbors?
5. Does it do this in the appropriate situations?
6. What happens, if a neighbor sends invalid or improper formatted information?
7. How does a router process valid information from is neighbors?
8. Is the routing table calculated correctly?
9. Does a router properly forward packets after the routing table has been updated?

Depending on the particular routing protocol to be tested, the fault models have to be refined and more specific test purposes have to be identified accordingly.

4.2. Open Shortest Path First Overview

Open Shortest Path First (OSPF) is a link-state intra-domain routing protocol and is the interior gateway protocol which has been recommended by the *Internet Engineering Task Force*[1] (IETF). The most recent specification of OSPF version 2 is provided in RFC 2328 [Moy98]. OSPF runs directly over IP using protocol number 89. This Section provides a general overview of how OSPF works.

For the participation in a link-state routing protocol, each router must have knowledge about the whole topology of the network. This information is exchanged with directly attached neighbors and is distributed over the whole network by a flooding procedure. But before any routing information or data traffic can be sent over a link, an OSPF-speaking router must assure, that the

[1] http://www.ietf.org

neighbor on the other side of the link is reachable. The OSPF *Hello Protocol* helps the router doing this. *Hello Packets* are sent periodically out of all interfaces on which OSPF is enabled. Routers, which share a common data link and which agree on certain parameters in these Hello Packets (like network mask, time-out intervals, etc.), are called *neighbors*. Each Hello Packet contains a list of all neighbors, from which the router has received Hello Packets recently. If a router receives a Hello Packet, and sees itself listed in the neighbor list, it is sure, that bi-directional communication between the two routers is possible[2].

When two routers have found out, that the communication link between them is bi-directional, they can form an *adjacency*. Adjacencies are special forms of neighborship, which are formed between two neighbors, that decide to exchange routing information. These adjacencies are not formed between every two neighbors, which have a bi-directional communication link between them. The forming of an adjacency depends on the network- and router-types (see Section 4.2.1). All the information about the network topology is kept in the *link state database*. When the routers decide to form an adjacency, they must first synchronize their link-state databases, which means that they exchange their current view of the topology of the network. Once this is done, the adjacency can be used in the process of flooding routing information all over the network.

OSPF knows a variety of different kinds of routing information. The individual pieces of information are called *link state advertisements* (LSAs). Each router describes itself and parts of the topology, dependent of its role in the network. Everytimes, a router detects a change in the topology, it must update the LSAs it is responsible for and send them to its adjacent neighbors. Additionally everytimes a router receives new routing information over one of its adjacencies, it must update its link state database with this information to stay synchronized with this router. In order to keep synchronized with all the other adjacent neighbors, the router must also send the new information to them, so they can update their databases, too. This process is called *reliable flooding* and is further described in Section 4.4.2.

A router's link state database contains information about the whole topology of the network. From this information, each router can calculate the shortest paths to all destinations and so build its routing table. The calculation of the shortest paths is done every time the link state database has changed, using Dijkstra's Shortest Path First (SPF) algorithm.

To reduce the size of link state databases and the amount of routing protocol traffic, OSPF introduces a hierarchical concept. Routers can be grouped in *areas*. A special area – the so called *backbone area* – interconnects all the other areas. Each of the areas has its own link state database. All routers in the area only have detailed knowledge about the topology of the area. Routes to destinations external to the area are summarized. A detailed description of the area concept is given in Section 4.2.2.

[2] Obviously, the other router has received a Hello Packet from the router recently and included the router on its neighbor list.

4.2.1. Network types

OSPF knows five different network types, which influence the process of building adjacencies. *Point-to-point* networks interconnect exactly two routers (e.g. serial lines). Routers, which are interconnected by a point-to-point network always become adjacent.

Broadcast networks connect more than two devices (e.g. Ethernet). Each packet, which is broadcast to the network can be received by all other devices. To minimize the number of adjacencies and to reduce routing protocol traffic on those networks, the concept of *designated routers* is introduced. The designated router (DR) maintains adjacencies with all other routers on the broadcast networks and controls the flooding procedure. Additionally, it originates an LSA representing the broadcast network to the rest of the area. Figure 4.1 illustrates the concept of the DR. To be able to recover quickly from failures of the designated router, a *backup designated router* (BDR) is introduced. The routers on the network form adjacencies not just with the DR but also with the BDR. So, if the DR fails, the BDR can take over its responsabilities in a minimum amount of time. The DR and BDR for a network are determined in a process called *DR election* (see Section 4.3.1).

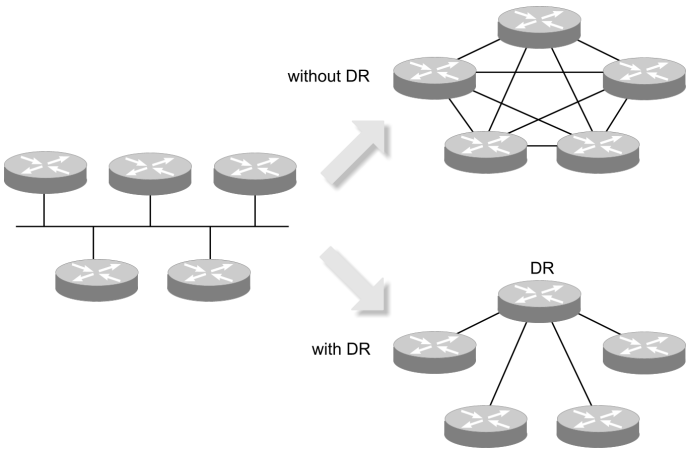

Figure 4.1.: Building of adjacencies with and without a DR

The *non-broadcast multi-access* (NBMA) network model is used for networks, that also connect more than two routers, but have no broadcast capability (e.g. ATM). On those networks, manual configuration of the neighbors is necessary, because they can not be detected automatically. Similar as in broadcast networks, a designated router and a backup designated router are elected in NBMA networks to reduce the number of adjacencies.

The *point-to-multipoint* network model can be used as an alternative to the NBMA model. Although this model is less effcient than NBMA, it is more stable. When running in this mode, all routers on the network are treated as if they were pairwise interconnected by point-to-point links. As a consequence, there is no designated router on these networks.

The fifth network type are *virtual links*, which are described later (see Section 4.2.2). Virtual links are special configurations, which are interpreted as point-to-point links.

The addressing of OSPF packets is dependent of the network type as well. On point-to-point networks, packets are always sent to the multicast address *AllSPFRouters* (224.0.0.5). On all other network types, most of the packets are sent as unicasts directly to the neighbor. The only exception to this are packets sent on broadcast networks, where LSAs are flooded to one of the multicast addresses *AllSPFRouters* or *AllDRouters* (depends of if the sending router is the DR or BDR). Acknowledgements are also sent to these multicast addresses. Retransmissions of LSAs are always sent as unicasts.

4.2.2. Areas

The size of the link state database grows with the size of the network. Maintaining a large link state database puts high demands on the CPU and memory requirements of a router. Additionally, the flooding of link state updates can contribute a remarkable portion of the traffic in a large network. To reduce the size of the link state databases, the amount of routing protocol traffic and the requirements on CPU and memory, a level of hierarchy was introduced in OSPF. An autonomous system can be divided into several logical groups of routers and links. These logical groups are called *areas*. Each area has its own link state database. Routers only have detailed knowledge about the topology of the areas, they belong to. Flooding is mostly limited to the area. Only summarized information is flooded into other areas. Each area has an area ID, which identifies the area in the autonomous system. It is usually formatted like an IP address.

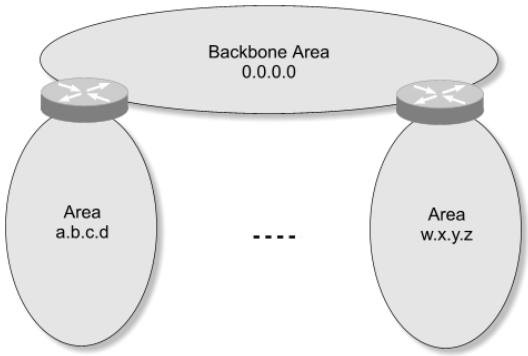

Figure 4.2.: Dividing an AS into areas

The top level area always has area number 0.0.0.0 and is called the *backbone area*. All other areas must be connected to that backbone area. Each router, which has interfaces to multiple areas must also have an interface to the backbone. These routers are called *area border routers* (ABRs). When sending routes from an area into the backbone, they are first summarized if possible (e.g. 10.0.0.0/24 and 10.0.1.0/24 could be summarized as 10.0.0.0/23). From

the backbone, these summaries are now flooded into the other areas. Since all routers which are connected to two or more areas are also connected to the backbone (i.e. are ABRs), all traffic between the areas must be routed over an ABR. Direct connections between two routers which are not connected to the backbone are not allowed.

All routers maintain one link state database for each area, they are connected to. Routers having only interfaces to one area are called *internal routers*. All routers having at least one interface in the backbone area are called *backbone routers*. Not all backbone routers necessarily have to be ABRs, because there may also be routers which have only interfaces in the backbone. In fact, every ABR is a backbone router. Routers which exchange routing information with other autonomous systems (possibly running an additional EGP, such as BGP4) are called *autonomous system boundary routers* (ASBR). These routers serve as gateways for routing data to external destinations. In order to make these external destinations reachable from within the whole AS, an ASBR advertises them throughout the autonomous system. Each internal router, ABR or backbone router may also be an ASBR.

Stub Areas

A considerable number of LSAs contained in an area's link state database may describe AS external destinations (see Section 4.5.7). When the exit point of an area needs not to be chosen on a per-external destination basis, this area can be configured as a *stub area*. Since descriptions for external destinations are not flooded into stub areas, the number of LSAs in the link state database can be reduced significantly. ABRs originate a default route into stub areas. When configuring an area as stub, some restrictions have to be considered. All routers in the area must know, that this area is a stub area. When two routers do not agree on this, no adjacency will be formed between them. Additionally, ASBRs cannot be placed in a stub area, since they announce external destinations, which cannot be flooded through stub areas.

Virtual Links

A *virtual link* is a link to the backbone through a non-backbone area. It can be used to connect an area to the backbone, which has no physical link to the backbone area. Virtual links are tunnels for routing protocol packets through the area. They are treated by OSPF like point-to-point links. The area, through which the virtual link is configured, is called the virtual link's *transit-area*. Virtual links always belong to the backbone and cannot be configured through a stub area.

4.2.3. Functional areas of OSPF

It can be seen easily, that the functionality of OSPF can be divided into several independent areas as proposed in Section 4.1.3. The detection and maintenance of neighbors is covered by the hello protocol. The exchange of link state databases and the flooding procedure control the sending and receipt of link

state advertisements. Dependent of the role of a router in the network it must originate different LSAs in different situations. Finally, the routing table is built using the information from the link state database, everytime the database is changed. In fact, these functional areas are not completely independent from each other. They rather build a hierarchy of functionalities, each depending on the underlying functionality. This must be taken into account, when a protocol implementation is tested.

The hello protocol builds the base for all other functionalities. When neighbors cannot be detected properly, no adjacencies will be formed and no routing information will be exchanged. So the hello protocol always has to be verified first. The next step is the building of adjacencies. During this step, the link state databases are synchronized. Only if this step is verified to work properly, reliable testing of the flooding procedure and calculation of routing tables can be made. The next step in the functional hierarchy is the flooding procedure. It must be assured, that a router properly floods all updates it receives and that it correctly originates new LSAs when appropriate. Once all these functionalities have been verified, the routing table calculation can be tested.

4.3. Neighbor Discovery and Maintenance

Each OSPF router must interact with its neighbors. To detect its neighbors, each router listens for incoming *hello packets* on all its interfaces. Each neighbor from which hello packets are received, is represented internally in a neighbor data structure, where the state of the communication with this neighbor is saved. In order to be detected by its neighbors, each router must send hello packets (see Appendix C.2) out of each interface. This is done periodically with an interval of *HelloInterval* seconds (e.g. 10s). In its hello packets, the router lists all the neighbors, it has discovered so far. Therefore, when a hello packet is received by a router containing its own router ID, the router can be sure that the communication link between itself and its neighbor is bi-directional.

Hello packets also serve as a means to detect failures of links and routers. Each time, a hello packet is received, an *inactivity timer* is restarted for this neighbor. The timer interval is known as *RouterDeadInterval* (e.g. 40s). When this timer expires, the neighbor is considered down and an adjacency with it is taken down. In addition, the router updates its link state advertisements and floods any changes over the network. The same thing happens, when a neighbor's hello packet does not contain the own router ID anymore. This means, that the neighbor cannot see this router anymore most probably caused by a link failure.

4.3.1. Election of the Designated Router

The hello protocol is also used for the election of the designated router and the backup designated router. Therefore, a router indicates its current opinion about which of its neighbors are DR and BDR in its hello packets. Each OSPF router has a *priority* associated with it, which influences the election procedure. Routers with a priority of 0 can never become DR or BDR. All routers with

a priority greater than 0 are called *eligible* routers. In the election procedure, a router now considers all its neighbors' opinions of which router is the DR and which router is the BDR. A router can determine the BDR and the DR as follows.

1. First, the BDR is elected. From the list of eligible routers, those routers are discarded, which have declared themselves to be DR. When one or more of the remaining routers have declared themselves to be the BDR, the one with the higest router priority is chosen as BDR. In a tie, the router IDs are compared and the router with the highest ID is selected. If no router was declaring itself to be the BDR, the router with the highest priority is chosen from the list of eligible routers. Again, in a tie, the router with the highest router ID is chosen.

2. After that, the router must elect the DR. From the list of eligible routers, those who declared themselves to be the DR are examined first. From these routers, the one with the highest priority is chosen. Again, the router ID breaks a tie. If no router claimed to be the DR, the newly elected BDR is chosen.

3. To ensure, that a router does not claim to be both DR and BDR, the previos steps must be repeated, if a router is no longer DR or BDR or if a router has been newly elected as DR or BDR.

A router which is connected to multiple broadcast or NBMA networks must not necessarily become DR on all these networks, when it becomes DR on one of them. DR and BDR are properies of router interfaces, not of routers. Each OSPF interface is controlled by the *interface state machine* [Doy98]. After the DR and the BDR have been elected, the calculating router itself is either DR, BDR or none of both. The interface state machine must now be brought into the appropriate state. Depending of its role on the network, the router must now also decide whether to build adjacencies with all or some of its neighbors.

4.4. Building Adjacencies and Flooding

Once bi-directional communication with a neighbor has been assured by the hello protocol, the routers must decide whether to become adjacent. This decision depends on the network-type. In point-to-point and point-to-multipoint networks and on virtual links, neighbors always try to become adjacent. In broadcast and NBMA networks, only the DR and the BDR become adjacent with each neighbor. The other routers only form adjacencies with the DR and the BDR in these neworks. Once the decision is made to become adjacent, the routers must exchange their link state databases. For this, each router first sends a description of its database to the neighbor. This one compares the descriptions with the contents of its own database. It then sends requests for those parts of the database it does not know, or for which the neighbor has more recent copies.

4.4.1. Database Exchange

The exchange of database descriptions is done by exchanging *database description packets* (see Appendix C.3). These contain summaries of each LSA from the router's database. The summaries contain only the headers of the LSA's (see Appendix C.7.1), which provide enough information to uniquely identify each LSA instance. The exchange procedure is controlled by one of the routers in order to make it reliable. Therefore, before the procedure begins, both routers must negotiate, who is the master. In order to do this, both routers send an empty database description packet in which they both claim to be the master. The sequence number of these packets is set to an arbitrary value. The router with the smaller router ID now becomes slave. It replies with a packet in which it does not claim to be the master anymore and in which the sequence number is set to the value from the master's initial database description packet. This packet is the first one which is filled with LSA headers.

Everytimes, the slave receives a database description packet from the master, it must acknowledge it by sending a packet with the same sequence number. If this acknowledging packet is not received by the master within the *RxmtInterval* (e.g. 5s), the master retransmits its last packet. The slave only sends database description packets in response to packets from the master. If the packet from the master has a new sequence number, the slave responds with a new packet containing the same sequence number. If the sequence number was the same as before, this must be a re-transmission from the master. That means that the master has not yet received an acknowledgement for this packet. The slave must acknowledge it again by retransmitting its last packet. The database description process is over, when both routers indicate in their packets, that they have no LSA headers left to send.

Requesting and Sending LSAs

During the exchange of database descriptions, LSA headers are received for every LSA that is contained in the neighbor's link state database. When a router finds a header for an LSA that is not contained in its own database or that is more recent than the copy in its own database, this LSA header is placed on the *link state request list*. The router now sends *link state request* packets (see Appendix C.4) containing all the headers from this list. Upon receipt of a link state request packet, the neighbor responds with *link state update packets* (see Appendix C.5) containing the requested LSAs. These LSAs are then deleted from the link state request list. When a requested LSA cannot be received within *RxmtInterval*, it is again requested from the neighbor.

Each received LSA must be acknowledged. When they are sent to a neighbor, the LSAs are set on the *link state retransmission list*. When an acknowledgement is received within *RxmtInterval*, the LSAs are deleted again from this list. If LSAs are not acknowledged within this interval, they are retransmitted to the neighbor. Acknowledgements can either be *explicit* or *implicit*. LSAs are explicitly acknowledged by sending a *link state acknowledgement packet* (see Appendix C.6), which contains the LSA header. Receiving a link state update

packet with the same instance of the LSA is an implicit acknowledgement for this LSA. When the Link state request list is empty, the process of building an adjacency with a neighbor is complete. The complete process of building an adjacency is controlled by the *neighbor state machine* [Doy98].

4.4.2. Reliable Flooding

To participate in a link state algorithm, each router must have knowledge about the topology of the whole network. This topology is described in the link state database. This database consists of a set of link state advertisements (LSA), each describing a certain part of the ropology. Every times, the topology changes (i.e. there is a change in the link state database), these changes must be sent to all other routers in the whole network, to assure, that their databases remain synchronized. This process is called *reliable flooding*. Everytimes an LSA is changed or a new one is received, it is sent to all adjacent neighbors in a link state update packet. These neighbors again send updates to all their adjacent neighbors, and so on. To make the process reliable, all LSAs must be acknowledged.

The flooding procedure depends on the type of network, the neighbors are connected to. On point-to-point networks, link state update packets are always multicast to the address *AllSPFRouters* (224.0.0.5). On point-to-multipoint networks and virtual links, they are unicast directly to the neighbor. On broadcast networks, the behaviour is slightly different. On these networks, those routers which are neither DR nor BDR, multicast updates to the address*AllDRouters* (224.0.0.6)[3]. The DR multicasts updates to all adjacent neighbors, using the address *AllSPFRouters*. The BDR does not flood any LSAs it hears from an adjacent router unless it sees, that the DR did not do so. On NBMA the behavior is similar to broadcast networks. Indeed, all packets are unicast, since these networks do not support multicast.

Like in the database exchange process (see Section 4.4.1), all LSAs which are sent to a neighbor are put on the link state retransmission list. When the LSA is not acknowledged within *RxmtInterval*, this LSA is retransmitted to the neighbor. Link state update packets which contain retransmissions are always unicast to the neighbor.

When receiving an LSA, a router must acknowledge it either explicitly or implicitly. Acknowledging an LSA explicitly means that the router sends a link state acknowledgement packet to the neighbor, which contains the LSA's header. An LSA can be implicitly acknowledged by sending a link state update packet to the neighbor, which contains the same instance of the LSA. Implicit acknowledgements can be used, if the router wanted to send an update to its neighbor anyway. Acknowledgements may either be sent delayed or directly. By delaying acknowledgements, several LSAs can be acknowledged in one single packet. On broadcast networks, even LSAs from multiple neighbors can be acknowledged in a single packet. To avoid unnecessary retransmissions, the

[3] *AllDRouters is the multicast IP address for the designated router and the backup designated router.*

period by which acknowledgements are delayed must be smaller than *RxmtInterval*. Direct acknowledgements are always unicast to the neighbor. These are sent every time, a duplicate LSA has been received from a neighbor. This indicates, that it has not yet received an acknowledgement.

When receiving an instance of an LSA, which is already contained in the router's database, the router must determine, which of the two instances is newer. This can be determined by three values contained in each LSA: the *sequence number*, the *checksum* and the *age*. The sequence number is a 32-bit signed number, ranging from *InitialSequenceNumer* (0x80000001) to *MaxSequenceNumber* (0x7FFFFFFF). When a router originates the first instance of an LSA, its sequence number is set to *InitialSequenceNumber*. Each following instance has the sequence number field incremented by 1. When *MaxSequenceNumber* is reached and a new LSA instance has to be originated, the present LSA instance has to be removed from all router's databases first. This is done by prematurely aging this LSA (see Section 4.5.2) and then originating a new LSA with sequence number *InitialSequenceNumber*. The checksum is a 16-bit unsigned number, which is calculated using a Fletcher algorithm as specified in [McK84]. The checksum is calculated over the whole contents of the LSA, excluding the *age* field, because this field is modified during the flooding process. When an LSA is flooded, each router increments the LSA's age by *InfTransDelay*, when sending it out of an interface. Given two instances of an LSA, the most recent one can be determined as follows:

1. Compare the sequence numbers of the LSAs. The one with the newest sequence number is the most recent LSA.

2. If the sequence numbers are equal, the checksums are compared. The LSA with the highest checksum is the most recent.

3. If the checksums are also equal, the age field is compared. If one of the LSAs has an age of *MaxAge*, it is considered the most recent. Receiving an LSA with the age set to *MaxAge* means, that it must be flushed from the database.

4. Else, if the ages of the LSAs differ by more than *MaxAgeDiff* (15 minutes), the one with the lower age is considered more recent.

5. Else, the LSAs are considered to be identical.

4.5. Link State Database

Link state advertisements, which describe the topology of an area are stored in the link state database. From this database, the routing table is calculated (see Section 4.6). Because each router participating in a link state protocol must have the same view on the topology, the maintenance of the link state database plays a central role in OSPF.

The link state database describes a directed graph, in which the vertices consist of routers and networks. Network vertices are used represent broadcast

and NBMA networks. Two router vertices are connected by an edge, when the routers have a connection via a physical point-to-point link or a point-to-multipoint network. An edge connecting a router to a network vertex indicates, that the router has an interface in this network. Networks can be classified as transit networks or as stub networks. A transit network is capable of carrying transit traffic and is specified by having both incoming and outgoing edges.

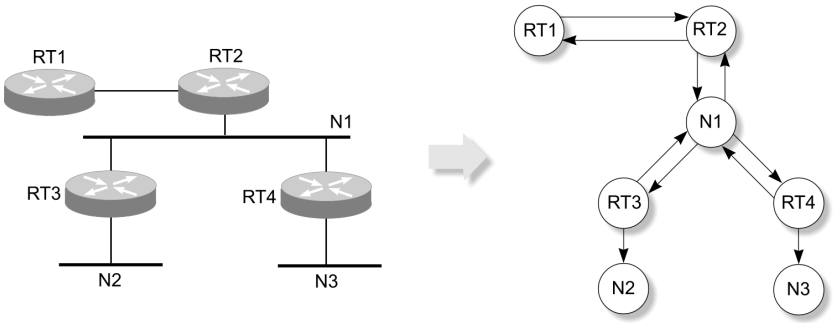

Figure 4.3.: Sample toplogy with the resulting link state database

Point-to-multipoint networks have no special representation in the link state database. OSPF treats these networks as if they consist only of point-to-point links between the routers and so represents them in the link state database. Each interface of an OSPF router has a cost associated with it. The edges in the graph are weighted with the interface output cost of the source vertex. Figure 4.3 shows a sample topology and the resulting Link State Database.

4.5.1. Aging the Link State Database

Each LSA has an age associated with it. This age is used to keep the link state database up-to-date. LSA's which exceed the *MaxAge* (1 hour) are reflooded and in that way deleted from the database. This assures, that there are no LSAs which are older than *MaxAge* within the whole network. The age of an LSA is set to zero, when the LSA is originated. Every router increments this age by *InfTransDelay*, when it floods the LSA on an interface. Additionally the age of an LSA is increased, when it resides in the link state database. When the age of a self-originated LSA exceeds *LSRefreshTime* (half an hour), a router originates a new instance of this LSA. This guarantees, that all LSAs are updated periodically. If a router's link state database becomes corrupted in some way, this assures, that it is updated with correct information periodically.

4.5.2. Premature Aging

When a router determines, that an LSA must be deleted from its database and the databases of all the other routers (e.g. due to a link failure, etc.), it can *prematurely age* this LSA. That means, that it sets the age of the LSA to *MaxAge* and floods the LSA. All routers, receiving an LSA with the age set to

MaxAge must delete it from their databases. Only the router which originated the LSA can prematurely age it.

4.5.3. Router LSA

The *router LSA* is originated by all routers. It describes all the router's links to a specific area. These can be links to other routers, transit networks, stub networks or virtual links. Router LSAs are only flooded within the area, the router belongs to. If the router belongs to multiple areas (i.e. it is an ABR), it originates one router LSA for each area.

4.5.4. Network LSA

Network LSAs are originated for each broadcast or NBMA network by the network's designated router. In the network LSA, all routers which are attached to the network are listed. Like router LSAs, network LSAs belong to an area and are therefore only flooded within this area.

4.5.5. Network Summary LSA

Network Summary LSAs are originated by area border routers to advertise IP networks into an area. These IP networks are destinations in other areas which are reachable by the ABR. Default routes can also be advertised as network summary LSAs. Like router- and network LSAs, network summary LSAs belong to a specific area.

4.5.6. ASBR Summary LSA

ASBR summary LSAs are also originated by ABRs. They are nearly identical to network summary LSAs but don't describe inter-area destinations. ASBR summary LSAs are used to advertise routes to autonomous system boundary routers (ASBR) into an area. The destinations advertised in ASBR Summary LSAs are always host addresses of the ABSRs.

4.5.7. AS External LSA

ASBRs advertise the external destinations they can reach in *AS external LSAs*. These LSAs are not associated with an area and flooded through the whole AS (excluding stub areas, see Section 4.2.2). AS external LSAs can also be used to advertise external default routes. AS external LSAs can specify two types of metric, which influence the routing table calculation.

4.6. Calculation of the Routing Table

The routing table is calculated using *Dijkstra's Shortest Path First* algorithm [CLR90] or any other algorithm that produces an equivalent output. The algorithm takes the router's link state database as input and produces a *shortest path tree* as output. From this shortest path tree, the routing table is derived.

The calculation of the shortest path tree is divided into several subsequent stages:

1. Calculate the shotest path tree for the area using the shortest path first algorithm, considering only routers and transit networks.

2. Add stub networks to the shortest path tree.

3. Calculate the inter-area routes and add them to the shortest path tree.

4. Examine transit areas' summary LSAs.

5. Add AS external routes to the tree.

4.6.1. Calculation of the shortest path tree

In this stage, the shortest path tree for an area is built using the Dijkstra Shortest Path First algorithm. Only the links between routers and transit networks contained in this area's link state database are taken into account. The *list of candidate vertices* contains those vertices for which paths have already been found. These paths are not necessarily the shortest paths. At any time during the execution of the Dijkstra algorithm, the shortest path tree contains only those vertices for which shortest paths from the root have already been found. Each vertex has an LSA associated with it (i.e. a router LSA or a network LSA), which describes all links to adjacent vertices (i.e. edges of the graph).

In the beginning of the algorithm's execution, the shortest path tree is initialized with the vertex representing the calculating router as the root. The list of candidate vertices is cleared. The algorithm adds one vertex to the shortest path tree in every iteration. Let V be the vertex, added to the tree in the previous iteration. Now V's links are examined. Each link connects V to another vertex W. If W is a transit vertex (i.e. transit network or router), which is not yet contained in the shortest path tree, the cost for reaching W from the root must be examined. If this cost is less than or equal to the value that already appears in the candidate list for reaching W, the existing entry for W is updated. Therefore the set of next hops for reaching W must be calculated (see Section 4.6.6).

If the candidate list is empty at this step, the algorithm terminates and the next step in the routing table calculation is performed (Section 4.6.2). Otherwise, the vertex from the candidate list is selected, which has the smallest cost. This vertex is deleted from the list and added to the shortest path tree. If there are multiple vertices on the list with the same least cost, an arbitrary one can be chosen. It must only be assured, that network vertices must be added before router vertices in order to find all equal-cost paths. After the vertex was added to the tree, a routing table entry for the vertex is created and the algorithm continues with the next iteration.

4.6.2. Adding stub networks

After the shortest path tree has been calculated, the stub networks must be added to the tree. Therefore all router vertices have to be examined again. The routers which are not reachable are discarded. Each stub network link from the reachable routers' LSAs is examined. If the cost to reach a stub network is less than or equal to the cost of an existing routing table entry, or if such an entry does not exist, the routing table must again be updated. After all stub networks have been added, the calculation of all intra-area routes is complete.

4.6.3. Calculation of inter-area routes

Once, the intra-area routes have been calculated, the inter-area routes can be calculated and added to the routing table. For this, the area's summary LSAs are examined. On an ABR, only the backbone's summary LSAs are considered. Only those destinations that are not reachable by intra-area paths are taken into account. For each of the remaining LSAs, the routing table entry for the ABR or ASBR that originated the LSA must be looked up. The cost for reaching the destination contained in the LSA is calculated by adding the cost of the inter-area path to the advertising router and the cost announced in the LSA. When there is no routing table entry for this destination yet, or if the extisting entry has a higher cost, the routing table has to be updated.

4.6.4. Transit areas

After that, the transit areas of virtual links are examined. This step is only executed on ABRs, that are the endpoint of one or more virtual links. The summary LSAs of transit areas are examined to see, if they provide any cheaper paths to some destinations. For this, the routing table entries of destinations are looked up, that are contained in a transit area's summary LSAs. If the entry does not exist, or if the cost to reach the destination using the transit area is higher than the cost already contained in the routing table, this LSA is discarded. Else, the routing table has to be updated.

4.6.5. AS external routes

In the last step of the routing table calculation, AS external destinations are added to the routing table. Therefore, all AS external LSAs have to be examined. Now look up the routing table entries for the ASBR that advertised this destination[4]. Among these routes prefer intra-area routes, that do not use the backbone. From these prefered routes, select the one with the least cost. The cost to reach the external destination must be calculated according to the metric type specified in the LSA. If the metric is of type 1, the cost is the sum of this metric and the cost of the path to the ASBR. Else, the cost is simply the type 2 metric from the LSA. Now, the routing table is updated, dependent

[4]There may be multiple routing table entries. Possibly there is one entry for each attached area.

of the cost of an existing entry for the destination and the metric type of this LSA.

4.6.6. Next hop calculation

OSPF explicitly supports multiple least cost routes to a destination. Each routing table entry can have multiple next hops for a given destination. These next hops are calculated in the routing table calculation procedure. OSPF makes no assumptions about which next hop to use in the process of forwarding IP packets. This decision is left to the specific implementation of OSPF.

The next hop calculation always gets a destination vertex and its parent vertex from the shortest path tree as input. If there is at least one intervening router on the path from the root to the destination (i.e. if the parent vertex is a router vertex that is not the root or the parent vertex is a network vertex whose parent is not the root), the list of next hops is simply copied from the parent. Otherwise, there are two cases to consider. If the parent is the root, the destination is directly connected to one of the root's interfaces. In this case, the next hop interface is simply the interface connecting the root to this destination. Otherwise, the parent vertex is a network, whose parent is the root. The next hop interface is now the interface connecting the root to this network. During the calculation of the routing table, paths can be found that have the same cost as paths that are already contained in the rounting table. In this case, the next hops of this new path are simply added to the next hops already contained in the routing table.

4.7. How to test OSPF

Section 4.1.3 proposed to subdivide the problem of testing a routing protocol into several smaller testing problems. Section 4.2.3 showed, how such a division can be made for OSPF. The particular testing problems can now be refined and concrete test purposes can be formulated.

The first stage of testing must assure, that neighbor discovery and maintenance are implemented properly. Hello packets must be sent periodically and addressed correctly according to the network type. A router must react properly on incoming hello Ppackets, including the router ID of the sending router in its own hello packets. When there are no more hello packets from a neighbor, this must be detected and handled properly. Additonally, the election of the designated router and the backup designated router must be verified in this stage.

The second stage of testing an OSPF implementation includes, that a router properly decides whether to form an adjacency with a neighbor. After that, the router's link state databases have ro be exchanged. For this, first the master/slave negotiation has to be done properly. Now, it must be tested, that both – the master and the slave – correctly react on incoming packets from the other router. They must decide, whether the received packet is in order and send an appropriate packet in response. It must be assured, that the routers properly fill their link state request list with LSAs from the neighbor's

database, they do not know. After that, these LSAs have to be requested from the neighbor. This must respond with appropriate updates. When a neighbor does not respond within a certain amount of time, the requests have to be retransmitted. Besides the database exchange, the flooding procedure is another important part to test in this stage. It must be verified, that updates are sent to the appropriate neighbors using appropriate addressing, dependent of the network type. Every update must be acknowledged properly. If this is not done in time, updates must be retransmitted. To make sure, that routers properly detect, when to make changes, they must be able to decide which of two LSA instances is the newer one. For this, routers that originate an LSA must properly advance the sequence numbers.

The third stage of testing must assure, that a router properly originates LSAs of the different types and updates them properly, when changes are detected. This stage must also assure, that LSAs are properly aged as they are kept in the link state database. After *LsRefreshTime*, a router must update each of its self-originated LSAs, to make sure that all routers in the area are provided with correct information periodically. After an LSA reaches *MaxAge* it must be removed from the router's link state database.

The fourth stage in testing includes the verification of the routing table calculation. It must be validated, that the shortest path trees are computed correctly and the forwarding table is built properly. This includes the correct calculation of the next hops for every given destination. It must be verified, that intra-area paths are preferred over inter-area paths and inter-area paths are preferred over external routes.

5. OSPF Test Suites

Chapter 3 introduced different methods for conformance testing of a protocol. It was shown, that formal methods are often not suitable for complex protocols like OSPF. Informal methods have the disadvantage, that the results can neither be proven to be correct, nor to be complete. This leads to the question, how OSPF can be tested in practice. It shows, that most of the testing done by vendors of networking equipment is done using informal methods. Tests are performed in test labs using test networks. In addition there are some test devices, which can be used to simulate complex topologies and which automate testing without the need for having a complete test network.

5.1. Formal Methods and OSPF Testing

Section 3.7 ahowed, that most of the theoretical methods related to FSMs are not suitable for a complex distributed protocol like OSPF. Although some parts of OSPF are specified as finite state machines (the neighbor- and interface state machines), these are neither fully specified, nor strongly connected. Another problem is, that some important parts of OSPF (like routing table calculation) are not specified by the means of FSMs. Besides that, OSPF has several components, which are running concurrently (e.g. multiple the neighbor state machines). So, none of the methods based on an FSM representation seems to be suitable for OSPF.

Methods based on FDTs require a formal specification of the protocol. OSPF is an Internet standard. These standards are mostly specified, using natural language. Because of this, the OSPF specification has to be translated using some FDT first. Natural language provides an intuitive understanding of how a protocol should behave. But natural language often is not precise and leads to ambiguities in the protocol standard. So, in the translation process again ambiguities may arise. However, there is an approach to use a formal method for the generation of an OSPF test suite. This approach is discussed in the following section.

5.2. Concurrent-TTCN based test suite

Each OSPF router communicates with several other OSPF routers and can therefore be seen as an entity in a parallel system. Such concurrent behaviour can not be modeled with TTCN (see Section 3.2.4), which is used for the description of test cases in the OSI confomance testing methodology. Concurrent TTCN (see Section 3.5) is an extension to TTCN which was developed to model

concurrent test behaviors. So C-TTCN provides a means for the modeling of test cases for OSPF. In [BWC98] an approach to conformance testing of OSPF is presented, which uses C-TTCN for the description of test cases. Figure 5.1 shows the abstract test architecture used in this approach. Each OSPF implementation (IUT) interacts with its environment through a set of network interfaces. These are called *interaction points* (IP). The tester consists of several test components. The *parallel test components* (PTC) interact with the IUT through several *points of control and observation* (PCO). The PTCs are synchronized and controlled by a *main test component* (MTC). Additionally, the PTCs may communicate with each other directly.

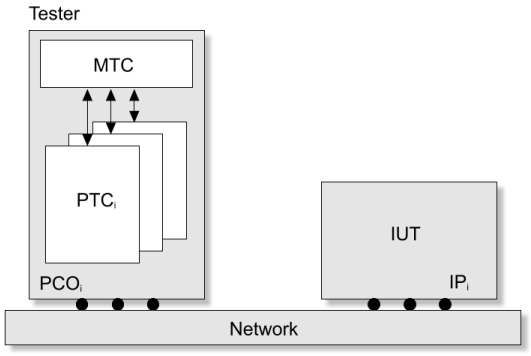

Figure 5.1.: Test architecture

To be able to use some formal method for the selection of test cases, the protocol has to be specified, using some formal description technique. Since the OSPF specification is written in natural language, it has to be translated first. For this, a description model called *Concurrent External Behavior Expressions* (CEBE) is proposed in [BWC98]. CEBE is an extension of Labeled Transition Systems, presented in Section 3.4.5. Specifications, using FDTs, such as LOTOS or Estelle can be represented as an LTS. Therefore, a CEBE specification may also be obtained using such a specification languages. To realize the test system presented in [BWC98], the natural language specificaion of OSPF had to be translated into a CEBE specification first.

CEBE can describe both the control flow and the data flow in a distributed system from an external viewpoint. So, the states of the CEBE specification are not equivalent to internal states of the OSPF specification. Because of this, some internal states of OSPF, that cannot be observed externally, may be omitted from the CEBE specification. From the CEBE specification, valid sequences of transitions can now be obtained. Based on these sequences, test cases can be derived, which are translated into C-TTCN.

The resulting test suite consists of over 4000 test cases. An analysis of the test suite showed, that some of these test cases are unimportant or have no clear test purposes. After the elimination of these test cases, there were 543 left. [BWC98] shows, that it is possible to apply formal methods to conformance testing of OSPF. Indeed, no statement about the coverage of the resulting test suite is made. There are two crucial parts, which influence the coverage. The

algorithm that is used to derive the test cases from the CEBE specification and the translation of the OSPF specification into CEBE. The latter of them cannot be formally proven to be correct.

5.3. InterOperability Lab Test Suite

Another OSPF test suite is provided by the InterOperability Laboratory (IOL) at the University of New Hampshire, which provides testing services for vendors of computer communications devices. Vendors with common interests form consortiums within the IOL, where they cover the costs of developing and performing tests and provide equipment which can be used to perform interoperability testing. Currently, there are 17 different consortiums like the Routing Consortium, the IPv6 consortium and others. Some prominent menbers of the IOL are 3Com, Cisco, Nortel Networks, Microsoft and many others. The IOL Routing Consortium has developed conformance test suites for various routing protcols, such as RIP, OSPF, BGP and others. In addition, there is a dedicated autonomous system for interoperability testing. This AS consists of routers from different vendors.

The conformance test suites provided by the IOL are not developed, using formal methods. Their test cases are derived directly from the specification via careful reading and analyzing the protocol standards. Step by step, the test suites are refined. Since many vendors use the test suites, they are improved and completed continuously. The test suites can neither be proven to be correct nor to be complete. But their impact on practical testing shows, that they provide a high degree of certainty, that an implementation that passes the tests, implements the protocol standard correctly and is able to interoperate with other implementations.

The IOL test suite for OSPF [Int00] consists of 78 test cases, which may again be divided into several independent test steps. The tests are divided into five categories. These are exactly the four functional areas of OSPF presented in Section 4.2.3 together with an additional category of tests.

- *Hello protocol* tests are responsible for the verification of the OSPF hello protocol. This includes tests, which cover establishment and maintenance of neighbor relationships as well as the designated router election in broadcast networks.
- *Flooding and Adjacency* tests cover the synchronization of link state databases and the reliable flooding mechanism of OSPF.
- *Link State Advertisement* tests verify, that link state advertisements are originated and received correctly by a router.
- *Route Calculation* tests verify that the routing table is built correctly from the link state database.
- *Configuration and Formating* tests finally verify, that necessary parameters can be configured and that OSPF packets are properly formated.

Each test case has a clear *test purpose*. This is a short statement, that describes the intention of this test. References are given to the parts of the RFC, where the acording behavior was specified. After that the relevant parts of the specification are discussed in detail with respect to testing. A *test setup* specifies the topology, which is used for testing together with basic configurations, such as link costs or virtual links. Finally the *test procedure* describes in detail, how the test has to be performed and which results should be observed.

OSPF distinguishes different network types (see Section 4.2.1). The addressing of OSPF packets and the decision, whether to build adjacencies depend on these network types. Indeed, the IOL test suite discusses all these network types, but the test setups contain only broadcast networks and virtual links. As shown in Section 4.6, the calculation of the routing table is done in several stages. First, the shortest path tree is built for the destinations internal to an area. Then inter-area and AS-external routes are added. The IOL test suite includes tests to verify, if inter- and intra-area routes are chosen properly and AS-external routes are added correctly. There are no explicit tests for the basic calculation of the shortest path tree, using only intra-area routes. To some extent, this is tested implicitly by the route calculation tests, but more extensive tests are missing.

5.4. Commercial Test Products

In addition to lab testing (e.g. at the IOL), vendors can use a variety of commercial products to perform tests of their protocol implementation. Many of these test products for OSPF can be used for performance and stress tests, but most of them do not cover conformance testing. So these are not considered here. Two commercial conformance test products for OSPF are presented below.

5.4.1. Automated Network Validation Library

The *Automated Network Validation Library* (ANVL) is a software test product, which is available from Empirix[1]. ANVL is a tool for conformance- and performance testing of protocol implementations running on network devices [Emp01b]. Test suites for many protocols are available, among them conformance and performance test suites for the routing protocols RIP, IS-IS, BGP4 and also OSPFv2. Additionally, toolkits for the development of own test suites are available. ANVL is used in testing of products by many well-known vendors like Cisco, 3Com, Nortel, Lucent and many others.

The ANVL software runs on Sparc Workstations running Solaris or on PCs running Linux or Windows NT. It tests a device by sending packets, receiving packets sent in response and analyzing this response. This can be done using multiple interfaces, connected to the implementation under test (IUT). ANVL can even simulate whole networks to test how an implementation behaves in different topologies (see Figure 5.2).

[1] http://www.empirix.com

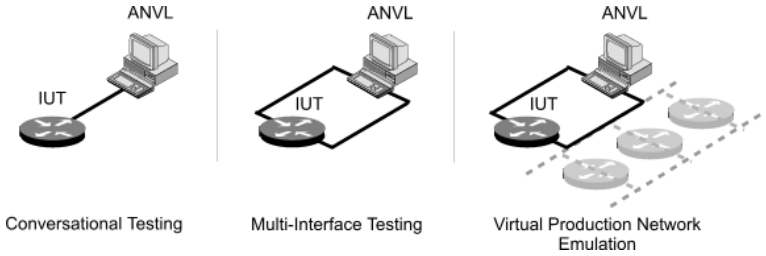

Figure 5.2.: Testing a router with ANVL

ANVL test suites are not developed using formal methods. From the standard, a *test asserion list* (TAL) is derived [Emp00]. This TAL summarizes the standard, making statements about what an implementation must or must not do. From the TAL, the test cases are then derived. Each test case verifies one or more test assertions. The ANVL OSPF Test Suite [Emp01a] consists of over 300 test cases, which cover both RFC 2328 and RFC 1583. The tests are divided into 12 groups.

- link state database
- functional summary
- bringing up adjacencies
- protocol packet processing
- the interface data structure
- the neighbor data structure
- link state advertisements
- flooding
- virtual links
- routing table calculation
- OSPF data formats
- LSA formats

The test suite originally covered only about 50 % of the protocol standard. Feedback from customers was used to improve the coverage. According to Empirix, the test suite now covers the complete OSPF standard.

5.4.2. QARobot / RouterTester

Another solution for OSPF conformance testing is provided by Agilent Technologies[2]. This solution consists of two hardware components: The *QARobot* is responsible for all basic conformance, stress and functional tests [Agi00]. The *RouterTester* adds wire-speed traffic generation capabilities and emulation of networks with several thousands of OSPF and BGP routers. An additional controller component runs the test suite software and provides one single user interface to the test engineer. Both, the QARobot and the RouterTester can be extended with different numbers of processors and test ports. There are test suites for different unicast and multicast routing protocols available, including RIP, IS-IS, BGP4 and OSPFv2.

The OSPF Conformance Test Suite [Agi01b] consists of more than 70 test cases which cover RFC 2328. Formal methods are not used in the design of test cases. Like in the IOL and ANVL test suites, the test cases are obtained by an analysis of the protocl standard. Like the ANVL test suite, the Agilent test suite can be customized. Test cases can be added or modified. The test suite is also divided into several groups:

- adjacency establishment
- adjacency maintenance
- adjacency deletion
- designated router election
- database synchronization
- preferred paths and hierarchical routing
- interface and neighbor state machines
- virtual links
- originating LSAs
- LSA fields
- router types
- network types

The network type tests include broadcast and point-to-point networks. NBMA and point-to-multipoint networks are not supported.

[2] http://www.agilent.com

5.5. Testing OSPF in Practice

The previous sections showed, that there are several different approaches to OSPF testing. Test suites can be derived formally or informally. Independent of that, the tests can be performed in a test lab, where the router has to interact with real network devices. In addition specialized test devices are available, which are connected to the device under test. These test devices perform the tests by simulating appropriate topologies. In practice most vendors do not use formal methods when testing their poducts [BWC98]. So there is no formal proof, that their protocol implementations comply to the specifications. However, most vendors use several different test suites, when testing their products.

5.5.1. Example: ZebOS Testing

The ZebOS Advanced Routing Suite is a suite of different routing protocols, containing implementations of BGP, OSPF and RIP. It is developed by IPInfusion[3] Inc. To test their products, IPInfusion performs extensive tests on its software [IP 01]. Testing is performed in several stages.

For *internal testing*, the ANVL conformance test suites are used. In addition, performance tests are performed, using other test tools. After the implementations have passed all tests, they are put into lab testing at the IOL and other test labs. There, they have to perform extensive conformance, reliability, stress and interoperability tests. Testing an implementation against multiple independent test suites increases the probability that offences against the specification are discovered.

[3] http://www.ipinfusion.com

6. Implementation

Testing plays an important role in a network simulator like SSFNet. Simulation results are only meaningful, if the simulation behaves exactly as the real world. So all protocols used in the simulator must be verified to be conformant to their respective specifications. The previous chapter showed different concepts and solutions for conformance testing of the OSPF protocol. This Chapter discusses, how these testing concepts can be realized in the SSFNet framework and presents the implementation of an OSPF test suite.

6.1. Conceptual Testing Framework for SSFnet

As mentioned before, it is crucial for a network simulator like SSFNet, that all implemented protocols comply to their respective specifications. Therefore, all protocols must be tested extensively. Since the most parts of SSFNet are regularly improved and completed, there is also a strong need for regression testing. This avoids that protocols which worked with one SSFNet version do not work any longer with a new release. Because of this, all protocols provided with SSFNet have a set of tests, which verify the behavior of these protocols.

The OSI testing methodology (see Section 3.2) provides concepts for conformance testing, which can be easily transferred into conformance testing within the SSFNet framework. Since the OSI methods are independent of any particular system, the test procedure (see Section 3.2.2) needs not to be modified. To be able to test distributed protocols such as OSPF, the coordinated test method can be extended to support multiple lower testers. It results in a test architecture, which is similar to the architecture used for testing with C-TTCN (see Section 3.5). This test architecture is depicted in Figure 6.1.

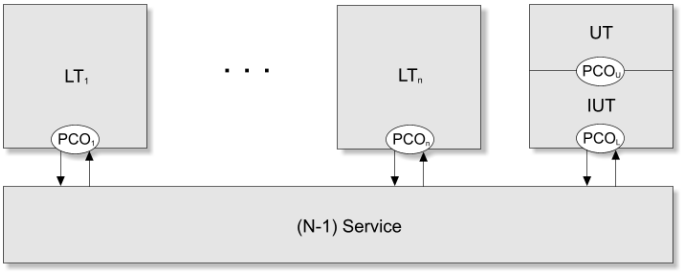

Figure 6.1.: Abstract test architecture for SSFNet

In SSFNet, the protocol implementation (IUT) is realized as a protocol session. The upper tester (UT) can also be realized as a protocol session in the

protocol graph of he IUT (e.g. the UT for a TCP session could be a HTTP session). The lower testers (LT) are placed in their own protocol graphs and can also be implemented in appropriate protocol sessions. The data exchanged between the testers and the IUT are protocol messages. The underlying service can be any lower level protocol that has already been implemented and tested in SSNet (e.g. IP, UDP or TCP). A closer look at the OSI test procedure shows, that there are four main phases:

1. Specification of test cases.
2. Implementation of an executable test suite.
3. Execution of the test suite.
4. Evaluation of test results and assignment of a verdict.

The first phase is completely independent of SSFNet, since it only depends on the protocol standard. The result is an abstract test suite, which must be implemented for the SSFNet environment. The upper and lower testers are used to bring the IUT from one state to another. This has to be done to verify, if an implementation correctly reacts on valid input. But also the correct reaction on invalid or erroneous input must be validated. So the testers must model the valid behaviors as well as the invalid ones. To model standard (valid) behavior, it is sufficient for many test scenarios to provide an appropriate DML scenario. But more advanced tests, which verify the behavior of an IUT under various error conditions, cannot always be modeled only by the means of DML. This strongly depends on the configuration possibilities of the IUT. Test behaviors, which cannot be modeled in DML, must be implemented in specialized testers. So the executable test suite will be a collection of DML scenarios and SSF models (e.g. implemented in Java) which model appropriate test behaviors.

In the OSI testing methodology, each test case has a preamble, which brings the IUT from an initial state into an appropriate state to run the test (e.g. establish a connection). After that, the test is run. Finally, there is a postamble, which brings the IUT back into the initial state. In the SSFNet framework, this postamble can be omitted, since each test case is executed in a different simulator run. That guarantees, that the IUT is definitely in the initial state, when a test begins.

The execution of the test suite will produce a set of test logs. Since the implementation is tested as a black box, the only source of information for the evaluation of the test results are the packets exchanged between the testers and the IUT, which can be observed at the PCOs. Therefore the testers must log these packets in an appropriate way (e.g. dump packets in *tcpdump* format). However, for some tests (e.g. verification of routing tables in routing protocols) it could be useful to have logs of internal datastructures as well. For this, the simulation log can be used as a source of information as well. Since the test suites should be independent of a specific implementation in order to reuse them for other implementations of the same protocol, this possibility should be sparely used.

After the execution of the test suite, the logs must be analyzed and a verdict must be assigned. This can either be done manually or automatically. When a user installs SSFNet on a machine, he might want to validate if SSFNet works properly in this environment. Since a user does not necessarily have knowledge on how to analyze the logs, the automatic evaluation of logs should be preferred.

6.2. The SSF.OS.OSPFv2 Package

The `SSF.OS.OSPFv2` package is currently under development at the Computer Networking Group at Saarland University. It is still in an early development stage but important features are already implemented[1]. `SSF.OS.OSPFv2` supports the hello protocol, database exchange, reliable flooding and routing table calculation. At the moment, only router-LSAs are implemented, but the other LSA types are planned to follow soon. The only network type available at the moment are point-to-point networks. This is sufficient, because most OSPF routers belong to the backbone of an AS[2], where point to point links are the most important link type. Nevertheless, the other network types will also be available in the future. In addition, features like virtual links and authentication are not yet available. These are not necessary for the general functionality of OSPF and will therefore be implemented later.

6.3. Test Suite Selection

The OSI framework for conformance testing makes no assumptions on how the test cases should be developed. Formal methods can be used as well as informal ones. Since the applicability of methods for a particular protocol may vary, the decision for a method must be made separately for each protocol. There are different criteria for the selection of a test selection method:

1. Correctness of the resulting test suite.
2. Coverage of the test suite.
3. Relevance and acceptance in practice.
4. Ease of implementation in SSFNet.
5. Applicability of the test suite during development.
6. Availability of the test suite.

Chapter 5 discussed several approaches to OSPF testing. For the implementation of an OSPF test suite for SSFNet, these approaches must be discussed with respect to the above criteria.

Formal methods provide provable coverage and correctness of the resulting test suite. Nevertheless, they are not very common in practical testing. Most

[1] The current release of SSF.OS.OSPFv2 is version 0.1.4.
[2] The backbone of an AS must not be mixed up with the OSPF backbone area.

test suites for OSPF conformance testing are designed by reading the standard and extracting the relevant parts. Although the test suites which are developed this way cannot be proven to cover the whole specification, they provide a sufficient coverage of the standard as well. Especially those test suites which are used by many vendors provide a good coverage, because they are used extensively. So the probability is high, that missing or incorrect tests are found soon.

To be able to package and distribute a the test suite with SSFNet, no external devices or additional software can be used. Since these systems are explicitly designed to test real devices, it would require some effort to integrate them into SSFNet – if this is possible at all. Another negative aspect of these commercial products is their price.

The IOL test suite description [Int00] is publicly available. All the tests of the IOL test suite consist of a test scenario description together with a test procedure. The test purposes and test procedures of the test cases are intuitively clear. DML configurations may be directly obtained from the scenario descriptions. That makes it easy to implement these tests in SSFNet. Since the single test cases are very simple, they can be used for testing during development as well. However, there are some features, that are not covered by the IOL test suite. There are only tests for broadcast networks – tests specific to point-to-point networks are not taken into account. For most of the test cases, this does not play a role, since the network type only affects the addressing of packets and the decision, when to build an adjacency. The tests for exactly these features have to be repeated, when broadcast networks are implemented. For the remaining tests, broadcast networks can be replaced by a set of point-to-point links (see Figure 6.2). Other tests, missing from the IOL test suite are tests for the simple calculation of intra area routes, using the Dijkstra algorithm (see Section 4.6.1). However, appropriate tests can be added easily to complete the test suite.

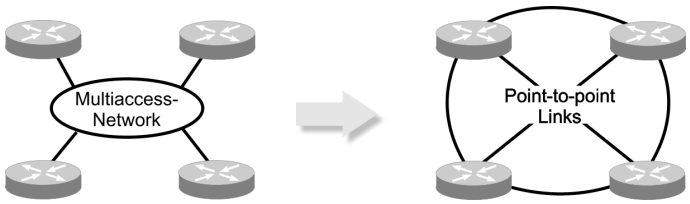

Figure 6.2.: Replacing broadcast networks

Although some tests are missing from the IOL test suite, it is a good basis for practical testing early in the development. It can be easily implemented in the SSFNet environment. Many vendors use the IOL test service – and with this the IOL test suite – when testing their products. This implies, that the IOL test suite provides a reasonable coverage and correctness for conformance testing. These considerations led to the decision to use the IOL test suite as a basis for testing the OSPF implementation in SSFNet. Whenever necessary, additions or modifications are made to meet the specific requirements in the

SSFNet environment.

6.4. Design of the OSPF Testsuite

The IOL test suite can be implemented with regards to the concepts discussed in Section 6.1. For OSPF testing, an upper tester is not needed, since OSPF does not provide any service to an upper layer. There are multiple lower testers, depending of the test case. Each tester can be represented by a router in the DML configuration. These routers run different protocol sessions, dependent of their function during the test. Most testers run normal protocol sessions (e.g. IP, ICMP and OSPF) in order to model the behavior of a normal router. This is sufficient to test most parts of the OSPF specification. Nevertheless, some tests require a non-standard behavior to verify, that a router correctly reacts on invalid or erroneous packets. These behaviours must be modeled in specialized testers, which must extend or modify the behavior of the standard protocol sessions.

For the OSPF test suite, three classes of non-standard behavior are needed for testers. In general, all topologies are configured statically in SSFNet, which is sufficient for current applications. In order to test the behavior of OSPF when the topology changes (i.e. links fail, etc.), these topology changes must be simulated by a tester. The second class of test behavior is to introduce simple errors into all packets of the same type (e.g. modify the network mask in all hello packets). Such behavior can be used to test how OSPF deals with packets, that contain invalid values in some fields. The third class of behaviors is needed to model more complex tests. Some tests require the tester to be aware of the state of an adjacency and to modify only cetain packets or originate new ones dependent of this state (e.g. modify the sequence number of the fifth database description packet). For some test cases it may also be necessary to combine these behaviors. So, all the test behaviors are implemented in appropriate protocol sessions, which can be combined in a protocol graph. All the protocol sessions for the testers are implemented in the `SSF.OS.OSPFv2.test` package (see Section 6.5 for a detailed description).

6.4.1. Structure of the Test Suite

As seen in Chapter 4, the OSPF functionality can be divided into several areas, which are mostly independent of each other. These areas are the *hello protocol*, *database exchange and flooding*, *origination of different LSAs* and *routing calculations*. Additionally, *general configuration and packet formatting* issues can be seen as a separate group. The IOL test suite is divided in exactly these groups. So the test suite can be seen as a collection of five subsidiary test suites, which can be executed mostly separate from the others. Each of these five test suites again consists of several test cases.

6.4.2. Structure of the Test Cases

Each of the test cases consists of a *test setup* and a *test procedure*. The test setup describes the network topology for this test. The test procedure describes in detail, how the routers should behave and which results should be observed. A basic DML configuration can be directly derived from the test setup. A detailed configuration and required behavior of the components can be obtained from the test procedure. During the simulation of a test scenario, OSPF packets must be logged, where necessary. For the evaluation of the test results, these logs can then be analyzed and a verdict can be assigned. In summary, the execution of each test case is divided into two phases:

1. simulation of the scenario
2. evaluation the logfiles and assignment of a verdict

The evaluation of results is done automatically by a PERL script which is provided with every test case. These scripts extract the necessary information from the logfiles of a test scenario and generate a verdict out of this information.

Assigning Verdicts

There are three possible verdicts, which can be assigned to every test case. *PASS* means, that the implementation performed exactly as expected. *FAIL* is assigned, when there were offences against the specification. The *INCONCLUSIVE* verdict is assigned, if it is not clear if the implementation complies to the specification. This can be the case when log files are missing or some prerequisites for the test are not fulfilled (e.g. when testing sequence numbering in the database description process, but no database description packets are sent at all). The evaluation of logfiles strongly depends on the test cases, so every test case has its own evaluation script. For each of the test cases a file named `report.log` is produced, which contains information about which test steps passed or failed. This file also contains the final verdict for the test case. To be able to evaluate the reports automatically, the final verdict is contained in a line which is formatted as follows:

```
### VERDICT for <test case name>: <PASS|FAIL|INCONCLUSIVE> ###
```

When all test cases of a test suite have been executed, all the verdicts must be examined to assign a final verdict for this test suite. When all subsidiary test suites have been executed, the verdicts for the particular test suites are examined and a final conformance statement is produced. Another PERL script (`verdict.pl`) is used for the assignment of verdicts for a test suite.

`verdict.pl` takes the names of the test cases as parameters. The reports of all test cases specified in the command line are scanned. If this report exists, it is scanned for the above line, which contains the verdict for the test case. If the file does not exist or if it doesn't contain a line as specified above, the test case result is considered *INCONCLUSIVE*. The syntax of verdict.pl is as follows:

```
verdict.pl <test_suite_name> <report_file> <test1> ... <testn>
```

The final verdict for the test suite is generated as follows: If all test cases have passed, the whole test suite gets a *PASS* verdict. If one or more *FAIL* verdicts have been found, the whole test suite gets a *FAIL* verdict. If there are one or more *INCONCLUSIVE* verdicts, but no *FAIL* verdict, the whole test suite's verdict will be *INCONCUSIVE*. `verdict.pl` also generates an output, which contains a verdict line as specified above. This output can again be redirected into a file called `report.log`. So `verdict.pl` can also be used to produce the final conformance statement.

6.4.3. The Testsuite Dictionary

Many of the test scenarios require the same or similar configurations for some components. These common configurations of components are stored in the testsuite dictionary `testsuite.dml` to keep the test case DML configurations simple. The dictionary consists of several parts:

- The *interfaces* part defines some common interface types (e.g. 100MBit or 1GBit).
- The *protocols* part defines common configurations for `ProtocolSessions`.
- Common router configurations are contained in the *routers* section.
- The *networks* part finally contains several networks of different sizes.

In the routers section there are common router configurations with different numbers of interfaces running different protocol configurations. Some test scenarios of the IOL test suite require the router to send a minimum number of LSAs. There are two approaches for realizing such scenarios. A tester could generate "virtual" LSAs to fill its link state database. In order to leave the OSPF session unchanged, this would require to simulate the complete building of an adjacency with a router connected to a virtual network. That means, that the tester has to implement parts of the OSPF protocol itself. The second approach to this problem uses the fact, that large networks can be generated easily in SSFNet (see Section 2.6). So, the router which has its link state database to be filled is simply connected to a network, in which a sufficient number of LSAs are generated (see Figure 6.3). In the test dictionary, there are networks with 8, 57 and 400 routers, which can be used for this purpose.

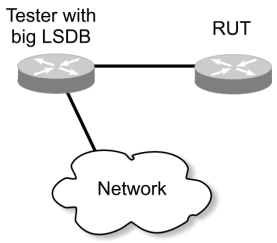

Figure 6.3.: Filling a router's link state database

6.5. The SSF.OS.OSPFv2.test Package

Some of the testers for the OSPF test suite cannot be modeled using standard SSFNet components. Normally, SSFnet is configured statically – that means, that the network configuration given in the DML file is maintained over the whole simulation. Wide parts of the OSPF specification deal with dynamic behavior of networks and how a router has to react on dynamic changes of the topology. To test those parts, this dynamic behavior has to be modeled in SSFNet. Five different classes of testers are needed to model all testers required by the OSPF test suite:

1. change a router's OSPF configuration at a given time during the simulation
2. shut down and restart an OSPF router at a given time
3. drop packets at given times to simulate link failures
4. modify certain fields in packet headers
5. specialized test behavior for particular tests, that may be dependent of the state of an adjacency

All these are features specific to the OSPF test suite and cannot be implemented using standard SSFNet models. Therefore these behaviors are implemented in the `SSF.OS.OSPFv2.test` package. An overview of the classes provided by this package is given in Table 6.1.

Class	**Description**
`Configurator`	Dynamically updates the configuration of an OSPF session.
`ConfigUpdateMessage`	Protocol message used to send updates to OSPF.
`Reset`	Resets an OSPF session at a given time.
`UnreliableIP`	Simulates links going down at a given time.
`IPwithErrorInjection`	Introduces simple errors into packets.
`PacketGenerator`	Models specialized complex testers for paticular tests.
`OSPFMonitor`	Dumps OSPF packets in tcpdump format.
`OSPFDumpPro`	Converts dumps from binary *tcpdump* format to text format.
`TOSDump`	Converts binary *tcpdump* files including the TOS fields.
`Logger`	Helper class to write output into the SSFnet logfile.

Table 6.1.: Package overview

Looking closer at the `UnreliableIP`, `IPwithErrorInjection` and `PacketGenerator` classes, it becomes clear, that they all have to manipulate packets sent to or coming from an OSPF session. The simplest way to realize this in SSFnet is to put a `ProtocolSession` between IP and OSPF, which manipulates the packets accordingly, drops packets or sends new ones. In practice, putting the tester between OSPF and IP does not work. OSPF sends its packets directly to the protocol with name "ip". So, in order to manipulate packets coming from OSPF, the tester must be included in the protocol graph, using the name "ip". The IP session then has to use another name, e.g. "ip_original". The `NIC` class, which models the network interfaces also sends packets to the class with name "ip". If the tester uses this name, all packets are given directly to the tester. The IP session does not receive any packets at all.

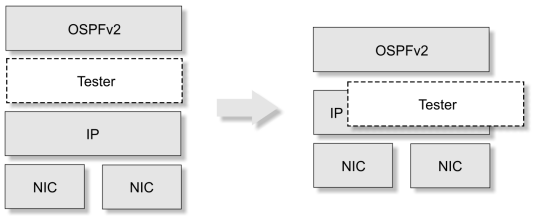

Figure 6.4.: Putting testers into a protocol graph

Because of this, the tester must additionally provide the functionality of IP. This can be realized by inheriance. The tester classes can simply extend the `IP` class, adding new functionality where necessary. To be able to use the functionality of different kinds of testers at the same time (e.g. link failures together with malformed hello packets), more complex tester classes again extend the simpler ones. This leads to the class hierarchy depicted in Figure 6.5.

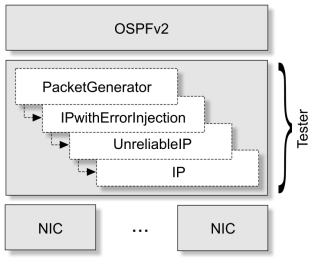

Figure 6.5.: Class hierarchy for testers

6.5.1. Unreliable Links

To model packet loss, the `NIC` class provides a facility to specify a percentage, at which incoming and outgoing packets are dropped. However, this feature does not allow a more accurate way of specifying link failures. The OSPF test suite requires a possibility, to specify accurately when and for how long

a link fails. For the OSPF test suite it is sufficient to drop all incoming and outgoing packets to simulate a link failure. This functionality is implemented in the `UnreliableIP` class. To model link failures in DML, this class has to be used instead of IP in a router's protocol graph. In the configuration of the `ProtocolSession`, link failures can be modeled as follows:

```
ProtocolSession [
    name ip
    use SSF.OS.OSPFv2.test.UnreliableIP
    fail [
        from      50.0
        until     100.0
        interface 0
    ]
]
```

Everytimes `UnreliableIP` receives a packet in its `push` method, it checks if the current simulation time lies in a configured failure period. If this is true, the interface associated with this packet is determined. If this associated interface corresponds to the interface specified in the configuraton, the packet is dropped. Else, the packet is passed on to IP's `push` method.

6.5.2. Introducing Simple Errors

For some tests, OSPF packets which contain errors have to be generated. Simply creating an appropriate OSPF packet and sending it without behaving like an OSPF router would not do the job. E.g. when the router does not send Hello packets there will never be an adjacency and no database description packets are exchanged. So, the tester can never send erroneous database description packets. A solution for this problem is to wait for packets from the OSPF session, which are sent to a neighbor. These packets can now be modified accordingly before they are sent. For this, the `IPwithErrorInjection` class is put into a `ProtocolGraph` as a replacement for IP. To be able to use the features of the `UnreliableIP` class, `IPwithErrorInjection` is derived from this class. The configuration of the tester's protocol session now looks as follows:

```
ProtocolSession [
    name ip
    use SSF.OS.OSPF.test.IPwithErrorInjection

    error [
        from      10.0
        until     20.0
        interface 0
        netmask   255.255.0.0
    ]
    error [
        from      20.0
        until     30.0
        interface 0
        bad_age   4000
    ]
    error [
        from      30.0
        until     40.0
        interface 0
        dd_mtu    2000
    ]
```

```
    fail [ ... ]
]
```

There are three kinds of errors, which can be introduced into OSPF packets at given times on certain interfaces. The *netmask* error is used to change the netmask field of hello packets to an arbitrary value. The MTU field of database description packets can be changed using the *dd_mtu* error. To change the age of LSAs in link state update packets, the *bad_age* error can be used. When a packet is sent to `IPwithErrorInjection`, it is first checked, if the packet has to be manipulated. If yes, the packet is manipulated in the appropriate way. After that, it is sent to `UnreliableIP` for further processing.

6.5.3. Complex Test Behavior

The `IPwitErrorInjection` class can be used to introduce simple errors into the OSPF protocol traffic. The class is not aware of the current state of an adjacency and simply introduces the same errors into all packets to which they are applicable. For some test scenarios a more advanced way was needed to manipulate OSPF packets or even introduce new ones. These features are implemented in the `PacketGenerator` class. The `PacketGenerator` keeps track of all OSPF packets. It maintains a data structure, where information about each neighbor and the associated adjacency is kept. Packets can now be originated or manipulated according to this information. The `PacketGenerator` is derived from the `IPwithErrorInjection` class, so this class's features are also available to the tester. The `PacketGenerator` can be configured from the DML file:

```
ProtocolSession [
    name ip
    use SSF.OS.OSPF.test.PacketGenerator

    test[
        interface 0
        behavior  <name of test behavior>
    ]
    error[ ... ]

    fail[ ... ]
]
```

Each test behavior is associated with an interface. The behavior of this interface is defined by a name, which corresponds to the name of the test case (see Appendix D for a list of test cases). Some test cases are subdivided into several independent test steps. Test behaviors for these test steps carry the name of the test case followed by a number, indicating the test step. Table 6.2 shows the test behaviors available in the `PacketGenerator` class. These behaviors are described in detail in Section 6.6.

6.5.4. Configuration Updates

Some test scenarios require dynamic updates of the OSPF configuration during a simulation. To configure such upates via the DML configuration, the

Test Name	Test purpose/Description
default	Normal OSPF behavior.
old_lsa_rcpt	Handling of the receipt of old LSAs (Test 2.6)
nbr_state_too_low	Correct handling of LSRequests from neighbors in a state < EXCHANGE (Test 2.7)
dd_retransmit1-4	Retransmission of database description packets (Test 2.8)
event_seq_number_mismatch1-8	Verify the generation of the SequenceNumberMismatch event (Test 2.9)
lsa_request_retransmit1-2	Retransmission of LSRequest packets (Test 2.17)
event_bad_ls_req1-2	Correct handling of the BadLSReq event (Test 2.18)
remove_lsa_from_retransmission_list	Removing LSAs from the retransmission list when they are discarded from the database (Test 2.21)
ls_sequence_number_wrap	Correct behavior when sequence number is wrapping (Test 3.22)

Table 6.2.: PacketGenerator behaviors

`Configurator` class was implemented as a `ProtocolSession`, which can be included into the protocol graph of the router whose OSPF configuration must be updated. The DML configuration for the configurator looks as follows:

```
ProtocolSession [
    name configurator
    use SSF.OS.OSPFv2.test.Configurator

    update [
        time 100.0

        [...OSPF configuration...]
    ]
]
```

Each update has a time associated with it and contains an OSPF DML configuration. This OSPF configuration may either be a complete configuration for all OSPF interfaces and areas or a partial configuration for a subset of interfaces or parameters. The parameters of the OSPF configuration which are not specified in the update remain unchanged.

The `Configurator` adds each update to a list and starts a timer for each update, which fires at the specified time. Once the timer has fired, the update is removed from the list and the OSPF configuration is packaged into a `ConfigUpdateMessage`, which is derived from the `ProtocolMessage` class. This message is sent to the OSPF session by calling its `push` method. Support for the handling of `ConfigUpdateMessage`s has been included in OSPF.

The class `com.renesys.raceway.DML.dmlConfig` provides an interface to the DML configuration. Each DML configuration represents a configuration tree – configuration updates for OSPF are a subtree of the global DML configuration. When sending updates to the OSPF session, these subtrees have to be copied into the `ConfigUpdateMessage`. The `dmlConfig` class is supposed to provide a method to copy a DML subtree to another place. But this method seems not to work correctly and delivers unpredictable results. Therefore the configuration is copied entry by entry.

6.5.5. Resetting OSPF

In some test cases, the OSPF session must be restarted during the simulation. To realize this, the `Reset` class was implemented which calls OSPF's `stop` and `init` methods at times specified in the DML configuration. To be able to configure the `Reset` class from DML, it was also derived from `ProtocolSession`. A typical DML configuration looks as follows:

```
ProtocolSession [
    name reset
    use SSF.OS.OSPFv2.test.Reset

    stop  10.0
    start 20.0
    stop  30.0
    start 40.0
    ...
]
```

6.5.6. Monitoring Packets

Whenever possible, the test cases should be implemented to be mostly independent from the particular OSPF implementation in order to be able to reuse the testsuite for other OSPF implementations. The implementation must be regarded as a black box. Tests are made by observing the inputs and outputs of the OSPF implementation. These inputs and outputs are typically IP packets, which must be logged. For this, SSFNet provides the possibility to define a `ProtocolMonitor` for an `IP` session. IP sends all packets to that monitor, before they are passed to the `NIC` or to a higher layer.

The `OSPFMonitor` class implements such a monitor, which logs all OSPF packets passing through an IP session in a binary *tcpdump* format. A utility to convert these binary files into an ASCII format is implemented in the `OSPFDumpPro` class. The generated output corresponds to the format generated by *tcpdump*[3]. To keep the generated logs small, not all IP headers are included in the converted files. One test requires the analysis of the IP TOS field. The `TOSDump` class can be used to generate a log file from the binary *tcpdump*, which contains only IP headers, including the TOS field.

[3]See http://www.tcpdump.org for details about *tcpdump*.

6.6. Implementation of Test Cases

This section describes how the test cases are implemented. Especially those test cases, which cannot be implemented using normal SSFNet classes and the tests requiring special or modified topologies are of interest here. Whenever possible the tests are implemented in a straightforward manner. That means, that a DML configuration is obtained from the test setup described in the IOL test suite. Detailed configuration of the particular routers is obtained from the test procedure description. For many test cases this is sufficient. The results can be obtained directly from the simulation logs. Because of the early development stage of OSPF (see Section 6.2) and the limitations of SSFNet (see Section 2.7) some tests cannot be performed yet or have to be modified to meet the current testing requirements. In particular, all broadcast networks are replaced by point-to-point networks. Features specific to broadcast networks (e.g. Designated Router election) cannot be tested. Another major difference between the current implementation and the OSPF standard concerns the use of mulicast. Multicast is used in OSPF to detect neighbors dynamically without having to configure each neighbor on every router. Additionally, multicast is used in broadcast networks to minimize the number of packets sent out of each interface. Since there are only point-to-point links in the SSFNet OSPF implementation at the moment, the only use of multicast would concern the automatic detection of neighbors. In the SSFNet environment, there is no need to configure neighbors manually, if multicast is not available. Each router has access to the complete network configuration in SSFNet. So, the neighbors can be obtained automatically from this configuration, without the use of multicast.

Since some features like virtual links or authentication are not needed in the OSPF implementation, these will not be implemented soon. Because of this it is not yet specified, how these features will be configured in DML. Therefore, the DML configurations for the according test cases can also not yet be prepared and are not considered in the following sections. However, no basic functionality of OSPF depends on these features.

6.6.1. Hello Protocol Tests

The hello protocol test suite consists of 12 test cases. They are used to verify, that the hello protocol works properly, neighbors are discovered and the DR and BDR are elected correctly. A complete overview of the test cases together with a short description can be found in Appendix D.1.

Most of the test cases for the Hello Protocol can be implemented with standard SSFNet components and the `UnreliableIP` class to take links up and down. The only exception to this is test 1.10, which verifies, that no adjacency can be formed, when the neighbors do not agree on certain parameters in their hello packets. One of these parameters is the network mask, which cannot be modified directly in SSFNet, since IP neworks (and the associated configuration) are assigned automatically. The `IPwithErrorInjection` class is used here to modify the *Netmask* field in all hello packets of one of the routers.

Test case 1.11 is a special case as well. When two routers do not agree about

the network they are connected to (i.e. the network masks are the same, but the network addresses indicate another network), no adjacency may be formed. Because of SSFNet's automatic IP address assignment, this class of errors will never occur. All interfaces connected to the same link are always configured properly. Therefore this test will pass implicitly.

6.6.2. Flooding and Adjacency Tests

The flooding and adjacency test suite consists of 21 test cases which verify, that the process of building adjacencies and the flooding procedure work properly. This includes testing, that the link state databases are exchanged correctly, when an adjacency is formed. In addition, the process of flooding LSAs throughout the network must be tested, including ackowledgements and retransmissions. A complete overview of the test cases together with a short description can be found in Appendix D.2. Again, many of the tests can be implemented using standard components of SSFNet and the `UnreliableIP` class to take links up and down. In addition, routers are restarted using the `Reset` class and receive configuration updates using the `Configurator` class. Other tests use the `IPwithErrorInjection` and `PacketGenerator` classes.

As discussed in Section 4.4, on multiaccess networks not every two routers form an adjacency. Test 2.1 is used to verify the correct building of adjacencies in these networks. Since multiaccess networks are not yet available, this test cannot be performed yet.

Test case 2.3 tests, how a router behaves, when the MTU field is not set correctly during the database description process. Since this value cannot be configured, it must be manipulated directly in the database description packets. For this, the `IPwithErrorInjection` class can be used.

When a router receives a self-originated LSA, that is newer than the instance of this LSA contained in its own link state database, it must originate a new instance of this LSA to keep the link state databases of all routers in the network consistent. This behavior is verified in test case 2.5. In order to do so, the router must be shut down, after it has synchronized its database with its neighbor. When the router is restarted it originates a new LSA, beginning with the *InitialSequenceNumber* (0x80000001). This is realized with the `Reset` class. To assure, that the LSA in the neighbor's database has a higher sequence number (indicating that it is newer), the link between both routers is taken down several times, everytime waiting for longer than *RouterDeadInterval*. This causes the adjacencies to be taken down and up again and assures, that new instances of the routers' LSAs are originated.

old_lsa_receipt

Test number 2.6 verifies, that a router properly handles the receipt of an LSA instance, that is older than an instance of this LSA that is already contained in its database. To test this, the tester must first flood an instance of the LSA having a new sequence number (e.g. 0x70000001). Then, after some time, the LSA is flooded again, having an older sequence number (e.g.

0x8FFFFFFE). This behavior is implemented in the `PacketGenerator` and can be activated using `old_lsa_rcpt` as the behavior name in the configuration of the `PacketGenerator`. To realize this test behavior, the `PacketGenerator` waits until the database exchange is over and the adjacency is established. Then it sends an update for its own router LSA, indicating a new instance of this LSA. After that, a timer is started, which fires after 10 seconds (enough time to have the neighbor install the new LSA in its database). When the timer fires, a new update is sent for the router LSA, which has an older sequence number. Since this behavior was specified differently in RFC 1583 and RFC 2328, the IOL test suite has different test steps which verify the behavior of a router against both specifications. However, the SSFNet implementation of OSPF only uses RFC 2328 as a reference. No downward compatibility is needed, so the tests for RFC 1583 can be omitted.

nbr_state_too_low

A router must discard link state requests and updates that are received before the master/slave negotiation for the database exchange process is over. This is tested in test case 2.7. To realize a tester, again an according bahavior of the `PacketGenerator` can be used (`nbr_state_too_low`). The `PacketGenerator` waits for the first database description packet from the neighbor and drops this packet. So the OSPF session does not know anything of the beginning database description process. Database description packets from the OSPF session are also dropped. When the first database description packet has been received by the `PacketGenerator`, a link state update packet containing the own router LSA is sent to the neighbor. After 10 seconds, a link state request packet is sent to the neighbor.

dd_retransmit

Test case 2.8 is used to verify, that database description packets are retransmitted when appropriate. Database description packets may only be retransmitted by the master. This test is divided into four independent parts, which test the correct behavior in different situations. The parts of this test case are implemented in a set of 4 test behaviors for the `PacketGenerator`.

The first part of the test verifies, that the slave does not retransmit database description packets. The tester must be the master for this test (i.e. it must have a higher Router ID). The `PacketGenerator` drops all database description packets, that are coming from the OSPF session, but the initial one. This behavior can be activated using the name `dd_retransmit1`.

The second part of this test case (`dd_retransmit2`) is used to assure, that the slave properly retransmits its previous database description packet, when it receives a retransmission from the master. For this, the link state database of the tester must be filled with enough LSAs to fill at least four database description packets. This is done by connecting a network to the tester, which contains enough routers (see Section 6.4.3). Again, the tester is the master in the database description process. The `PacketGenerator` now simply drops the

third database description packet coming from the RUT. After *RxmtInterval*, the OSPF session on the tester will retransmit its last packet to the RUT. This test implicitly verifies, that the master correctly retransmits database description packets after *RxmtInterval*, when it does not receive a packet from the slave.

An explicit test for this can be done, using the `dd_retransmit3` behavior. This time, the tester must be the slave. All database description packets from the master but the first two packets are dropped. So the OSPF session on the tester only receives these first two packets. Since the slave must not retransmit any database description packets unless it receives a retransmission from the master, it does not send any packets after that.

The fourth part of this test verifies, that the slave retains its last database description packet for *RouterDeadInterval* after it receives the final packet from the master. For this test, the tester must again be the master. The `PacketGenerator` waits for the last database description packet from the slave. Once, this packet is received, a timer is started, which expires after *RouterDeadInterval*. When the timer fires, the last database description packet is retransmitted to the slave. The slave must now retransmit its last packet in response. This test behavior is activated using the name `dd_retransmit4` in the DML configuration of the tester.

event_seq_number_mismatch

Test case 2.9 is used to verify, that a router reacts correctly on some errors during the database description process. The test is divided in eight independent test steps. Each test step can be implemented using a separate `PacketGenerator` behavior. Each of the test steps can be activated in the DML configuration using the name `event_seq_number_mismatch`, followed by the test step number (1-8).

When the options field in database description packets of a neighbor changes during the database description process, it must be aborted immediately and then be restarted. The `PacketGenerator` waits for the second database description packet from the neighbor. The options field of the database description packet, which is sent to the neighbor in response, is then manipulated.

The second test verifies, if a router correctly restarts the database exchange, when it receives an initial database description packet unexpectedly. For this, the `PacketGenerator` simply sets the *Initial*-bit in its third database description packet.

Whenever a database description packet is received during the database exchange process, which has a sequence number that is higher than expected, the database exchange must be restarted. To verify this, the `PacketGenerator` simply increases the sequence number of the third database description packet.

The fourth test step is used to verify, that a router correctly restarts he database exchange, when it receives a database description packet, which has the sequence number set too low. This is implemented in the `PacketGenerator` by decreasing the sequence number of the third database description packet.

The next test step verifies, that the database exchange is restarted, when

the master does not claim to be the master anymore during the database exchange. For this, the tester must be the master during database exchange (i.e. it must have the higher router ID). When the neighbor sends its first non-initial database description packet, the `PacketGenerator` simply clears the *Master*-bit in its next database description packet.

When a router receives a database description packet more than *RouterDeadInterval* after the final database description packet has been sent, it must restart the database exchange process. So, after the final packet is received, the `PacketGenerator` starts a timer, which waits for 50 seconds. When this timer fires, an additional database description Packet is sent to the neighbor.

The seventh test step verifies, that the database exchange is restarted, when a router receives an LSA header with an unknown LS type in a database description packet. For this, the `PacketGenerator` adds a new LSA header with LS type 0x7F in its second Database Description packet.

The last test step verifies, that the database exchange is restarted, when a router receives a header of an AS-external LSA in a stub area. AS-external LSAs are not yet implemented in OSPF. Because of this, the according test behavior for the `PacketGenerator` is not yet implemented.

lsa_request_retransmit

Test case 2.17 is used to verify, that a router properly retransmits link ltate lequest packets, when it does not receive the requested LSAs within *RxmtInterval*. This test consists of two independent test steps. Testers for these test steps can be realized using the `PacketGenerator`. In the DML configuration the name `lsa_request_retransmit` followed by the test step number (1-2) can be used to activate the appropriate test behavior for the tests.

The first test step verifies, that a router properly retransmits requests for LSAs, that are not received within *RxmtInterval*. For this, the tester is connected to a network with 57 routers in order to have its link state database filled. After that it is connected to another router. During the database exchange it sends several database description packets full of LSA headers. These LSAs are then requested by the neighbor, but no updates for these LSAs are sent to the neighbor.

When a router receives updates for some of the requested LSAs, it must not retransmit requests for these LSAs anymore. This is verified in the second test step. The `PacketGenerator` is again connected to a network in order to fill its link state database. Again, the router is then connected to another router, which requests all LSAs. The `PacketGennerator` now drops all link state update packets but the first one, so some of the requested LSAs are sent to the neighbor.

event_bad_ls_req

During the database exchange, a router must request all LSAs from its neighbor, that are not contained in its own link state database or for which it has older copies. Whenever a router does not behave like this, the database must be

restarted. Additionally, when a router receives an LSA that it has requested from its neighbor, but this LSA is older than an instance already contained in its link state database, the database exchange must be restarted. Test case 2.18 verifies this. To perform these tests, the `PacketGenerator` can be configured with the `event_bad_ls_req` behavior. Since this test consists of two independent test steps, the number of the according test step (1-2) must be appended to the behavior name in the DML configuration.

The first test step verifies, that the database exchange is restarted, when a neighbor requests an LSA that is not in the router's link state database. For this, the `PacketGenerator` adds a header for an imaginary LSA to the first link state request packet sent to its neighbor.

In the second test step it must be verified, that a router restarts the database exchange process, when it receives an LSA that it has requested from its neighbor, but which is older than an instance of the same LSA, that is already contained in the router's link state database. To test this, the `PacketGenerator` sends an update for its own Router LSA after the adjacency has been brought up. This LSA instance has sequence number 0x80000105. Then the OSPF session is shut down for longer than *RouterDeadInterval* to bring the adjacency down. When the OSPF session is restarted, the `PacketGenerator` indicates during the database description process, that its Router LSA has sequence number 0x80000205. This causes the neighbor to request this LSA. When the tester sends an update for the LSA, a sequence number smaller than 0x80000105 is used.

remove_lsa_from_retransmission_list

Test number 2.21 is used to verify, that a router properly removes LSAs from all its neighbors' retransmission lists, that are rmoved from its link state database. To realize this test, a router must originate two new instances of an LSA within *RouterDeadInterval*. On real routers, this ca be done by enabling and disabling an interface, which causes a new Router LSA to be originated immediately. In SSFNet, there is no mechanism to simulate this, since link failures are only detected after the *RouterDeadInterval* is over. So a `PacketGenerator` behavior must be used for this test (`remove_lsa_from_retransmission_list`). The test scenario is changed according to Figure 6.6.

Figure 6.6.: Modified test setup for test case 2.21

After all routes have synchronized their databases, the link between TR and RUT is taken down. The `PacketGenerator` on the tester now sends an update for its router LSA, including an imaginary stub network, so RUT begins retransmitting this update to TR. After 20 seconds, the tester again announces an update for its router LSA, this time without the imaginary stub network.

When RUT receives this update it must delete the old instance from its link state database.

6.6.3. Link State Advertisement Tests

The link state advertisements test suite has 22 test cases, which are used to verify that all kinds of LSAs are originated and handled correctly. A complete overview of the test cases together with a short description can be found in Appendix D.3.

Because of the early state of the OSPFv2 implementation, there are currently only router LSAs. Since most of the link state advertisement tests deal with the origination of different LSA types, most of these tests can not yet be performed. However, the DML configurations for these tests can be provided in a straightforward manner, using only standard SSFNet classes and the `UnreliableIP` class. Some of the test scenarios additionally require the use of the `Configurator` class to update link costs during the simulation. Since the test configurations are clear, the tests can be performed as soon as other LSA types become available. The tests dealing with router LSAs can also be performed only partially at the moment, since the only link types available at the moment are router links and stub network links. Again, the remaining tests can be performed as soon as the features are implemented in OSPF.

Tests number 3.13 and 3.14 are used to verify the correct use of area address ranges. Area address ranges can be used to help area border routers when originating summary LSAs. Multiple networks can so be combined and advertised together in a single summary LSA. Since SSFNet assigns IP addresses automatically, it is not easy to configure these address ranges from DML. Indeed, this problem will be solved, when summary LSAs are implemented. Until then, these tests can not yet be performed.

ls_sequence_number_wrap

Everytimes a router must increase the sequence number of a self originated LSA past *MaxSequenceNumber* (0x7FFFFFFF), it must delete this LSA from all routers' link state databases, before it can originate a new instance of this LSA with a sequence number set to *InitialSequenceNumber* (0x80000001). This is done by premature aing old LSA (see Section 4.5.2). Since there are over 4 billion sequence numbers in the range from the initial to the maximum sequence numbers, it would take a long time until the router must increment the sequence number of one of its LSAs past the maximum value. To shorten the simulation, a behavior is implemented in the `PacketGenerator`, which simplifies this (`ls_sequence_number_wrap`). During the database exchange, the `PacketGenerator` receives a router LSA from its neighbor. After the adjacency is established, it simply modifies the sequence number of this LSA to *MaxSequenceNumber* and floods it back to its neighbor. The neighbor must then originate a new router LSA, since it has received a self-originated LSA which is newer than the copy contained in its own database. In this new instance, it must increase the sequence number past the maximum value.

6.6.4. Route Calculation Tests

The IOL test suite provides 9 test cases to verify the calculation of the routing tables. A complete overview of the test cases together with a short description can be found in Appendix D.4.

The process of building the routing table is done in several stages (see Section 4.6). The first stage calculates the shortest path tree for an area using only router- and network-LSAs. The following stages add stub-networks, inter-area routes and AS-external routes. A closer look at the routing tests in the IOL test suite shows, that the basic calculation of the shortest path tree and the calculation of next-hops is not tested explicitly. Tests for these calclations have to be added, to assure, that a router properly calculates the routing table and the next-hop interfaces. A description of these additional tests can be found in Section 6.6.6.

In the later stages of the routing table calculation, summary-LSAs and AS-external LSAs are used. In the current release of OSPF these are not yet implemented. As a consequence, these stages of the routing table calculation are not implemented as well. As a consequence, the routing tests from the IOL test suite cannot be performed yet. Nevertheless, the DML configurations for these tests can be obtained from the test descriptions using the `UnreliableIP`, `Reset` and `Configurator` classes.

6.6.5. Configuration and Formatting Tests

The fifth subsidiary testsuite provides 14 test cases to verify that certain parameters are configurable and that OSPF packets are formatted correctly. A complete overview of the test cases together with a short description can be found in Appendix D.5.

Again, many of the tests verify things that are not yet implemented in the current version of OSPF. Other tests are not applicable in the SSFNet environment at all, because behavior that is tested in these test cases can never occur in the SSFNet implementation of OSPF. Tests 5.9 to 5.13 are used to verify different "length" fields, such as the packet length, the number of LSAs contained in a packet or the number of links in a router LSA. In OSPF implementations for real routers, packets are transmitted as arrays of bytes, which may be of variable length. The different length fields are needed to determine, where a packet ends. But in SSFNet, packets are not byte arrays. Packets are objects in SSFNet. So, there is no need to determine where the packet ends. Lists of LSAs or LSA headers are also not byte arrays in SSFNet. They are contained in the packtes as vectors of LSAs. The number of LSAs is not saved explicitly in the packets but can be obtained through the use of appropriate methods of the `Vector` class. For the test suite, this means, that the class of errors that is tested by these test cases will never occur in the SSFNet implementation. So these test cases can be omitted.

All OSPF packets should have the IP precedence field to *Internetwork Control* (0xC0). This is validated by test 5.4. SSFNet implements only a subset of the IP standard, which is sufficient to simulate most scenarios. The IP precedence

field is not implemented in SSFNet, so this test cannot be performed. Most of the other tests are used to test features of OSPF which are not yet implemented in the current version. This includes authentication and virtual links. These test cases cannot be performed yet.

Test number 5.8 verifies, if a router properly evaluates the checksums of LSAs. Since the checksum field is not yet provided by the LSA datastructure, a tester for this test case can not be implemented. Once checksums are implemented for LSAs, the `IPwithErrorInjection` class can be modified to support the manipulation of the checksums in all LSA headers in order to model this test behavior.

6.6.6. Additional Tests

As seen in Section 6.6.4, tests for the first stage of the routing table calculation are missing from the IOL test suite. However, it must be shown, that the Shortest Path First algorithm works properly. The algorithm cannot be proven to be implemented correctly simply by testing one scenario. Nevetheless, when the algorithm works properly for an appropriate scenario, there is some confidence that it works correctly for all scenarios. To find such a scenario, the structure of Dijkstra's algorithm must be clear. The algorithm contains a main loop, which terminates when there are no nodes left to be added to the tree. Termination of the algorithm is guaranteed, since each iteration of the loop adds exactly one node to the tree. It can be assumed, that the OSPF implementation of Dijkstra's algorithm terminates for all scenarios, when it does for an arbitrary one. To test if the calculated tree is really the correct tree of shortest paths, a similar approach can be followed. This approach does not prove the correctness of the algorithm, but if the test scenario is chosen carefully the chances are good, that the algorithm works for all scenarios.

Test Case 6.1 – spf_simple

This test case is used to verify the calculation of the shortest path tree and the set of next hops. The test setup together with the link costs is depicted in Figure 6.7. All link costs are chosen, so that no equal-cost routes occur. After all routers have synchronized their databases, all networks must be contained in the routing table of the RUT. The minimal cost and outgoing interfaces for each destination can now be calculated manually and compared to the entries in the routing table of the RUT.

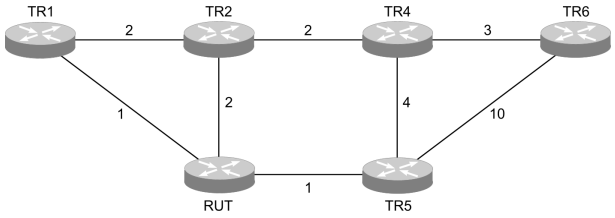

Figure 6.7.: Test 6.1 – Simple Shortest Path First calculation

Test Case 6.2 – equal_cost_routes

OSPF explicitly supports multiple least-cost routes to the same destination. This is considered during the next-hop calculation. The link costs of the scenario shown in Figure 6.8 are chosen, so that there are multiple least-cost routes to several destinations. Again, after the link state databases of all routers have synchronized, all destinations must be contained in the RUT's routing table. Again, the next-hops and the cost for each destination must be calculated manually and then be compared to the entries in the RUT's routing table.

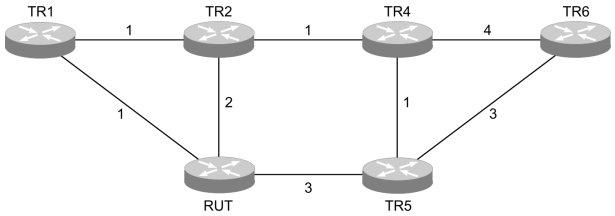

Figure 6.8.: Test 6.2 – Calculation of equal-cost routes

Test Case 6.3 – routes_use

This test case verifies, that the routes calculated by OSPF are correctly used, when a router forwards packets. For this, a simple TCP-client and an according server are connected to the network depicted in Figure 6.9. The client uses RUT as its default gateway, the server uses TR3 as its default gateway. Packets are captured on TR1. All packets from the TCP client and server must travel through TR1.

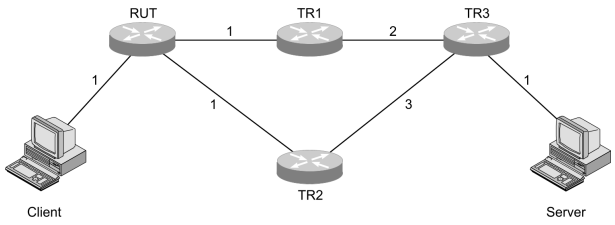

Figure 6.9.: Test 6.3 – Use of routes calculated by SPF

Test Case 6.4 – table_change

This test case verifies, that routing tables are changed correctly, when a link fails. For this, the topology from test case 6.2 (see Figure 6.8) can be used. After the link state databases of all routers have been synchronized, the link between RUT and TR1 is taken down using the `UnreliableIP` class. After *RouterDeadInterval*, the routers detect the failure and originate new LSAs. After the updates have been flooded throughout the network, the routing tables

must be checked. For comparison purposes, the new routing table can again be calculated manually.

6.7. Test Results

The execution of the test suite generates one of three possible verdicts for each test case. *PASSED* is assigned, when the OSPF implementation behaves exactly as required by the specification. *FAIL* is assigned, whenever an offence against the specification is found. *INCONCLUSIVE* is used for test cases, where no verdict can be assigned. This is the case for all tests cases which test parts of the specification which are not yet implemented. Table 6.3 shows an overview of test results for the current OSPF implementation version 0.1.4.

It can be seen easily, that most of the test cases are *INCONCLUSIVE* at the moment. This is not surprising since the OSPF implementation is still in a very early stage. The *PASS* verdicts indicate, that the according features have been implemented according to the RFC. The most interesting results are the test cases, which have *FAIL* verdicts. These are now discussed in detail.

Test 1.1 – hello_addressing_and_timing

This test case verifies, if hello packets are sent periodically and addressed to the multicast address *AllSPFRouters*. As discussed before (see Section 6.6), multicast is not yet available. OSPF packets are always unicast in the current OSPF version, which causes this test to fail.

Test 2.3 – database_description_mtu_match

Database description packets contain a MTU field, which must be set appropriately. When this field indicates a MTU that is higher than the MTU of the router's receiving interface, the database description packet must be dropped. Obviously, the OSPF implementation does not handle this field at the moment. Packets with an MTU higher than the router's own MTU are not dropped. Although this test indicates, that a part of the specification is not covered by the OSPF implementation, it has currently no effect on simulations in SSFNet. The MTU is dependent on the underlying link layer, which is not implemented in SSFNet at all. So OSPF assumes a MTU of 1500 for all interfaces.

Test 2.15 – lsa_multicast

In the flooding process, link state update packets must be sent to one of the two multicast addresses *AllSPFRouters* or *AllDRouters*, dependent of the network type. As dicussed for test case 1.1, multicast is not yet available, which causes this test to fail.

Test 2.18 – event_bad_ls_request

The process of building an adjacency must be restarted in two situations:

1. a neighbor requests an LSA that is not in the router's link state database or

2. an instance of an LSA received from a neighbor is on this neighbor's link state request list (which indicates, that the LSA has been requested from the neighbor), but the received instance is older than an instance of the same LSA, that is already contained in the router's database.

The latter of the two tests fails. When the router receives the update with the old LSA instance, it installs this LSA in its database. The database exchange is not restarted.

Test 2.21 – remove_lsa_from_retransmission_list

This test case shows the need for regression testing, since it passed for OSPF version 0.1.3. The test verifies, that LSAs are removed from all retransmission lists, when they are not in the router's link state database any longer. The test scenario is built, so that the RUT retransmits an updated LSA to a neighbor which is currently not reachable anymore. But the test logs show, that there are no retransmissions sent at all. Maybe, this is a bug, which was introduced through some changes in the flooding procedure.

Test 3.22 – ls_sequence_number_wrap

When the sequence number of an LSA reaches the value *MaxSequenceNumber* (0x7FFFFFFF), a router must first prematurely age this LSA, before it can originate a new instance of this LSA with sequence number *InitialSequenceNumber* (0x80000001). This test shows, that the router does not recognize that it has to handle this special case. The sequence number is simply increased to the value 0x80000000, which is not a valid sequence number at all.

Test 6.2 – equal_cost_routes

This additional test case is used to verify, that the set of next-hops is calculated correctly, when there are multiple least-cost routes to a destination. For the evaluation of the routing tables, the correct routing table is calculated manually. The test shows, that one route has an additional next hop, although the destination can only be reached at a higher cost, when this next-hop is used.

Test 6.3 – routes_use

This test case verifies, that the routing tables are activated correctly when they have been calculated. Therefore, a TCP connection is established between two hosts. According to the log, the client host sends lots of TCP SYN packets, but receives no response. A look at the simulation log shows, that IP is dropping packets due to invalid IP addresses. This indicates, that there is a problem in the OSPF/IP interaction.

Test 6.4 – table_change

Since this test case uses the same scenario as test number 6.2, it can be expected, that the same error occurs. Indeed, the additional next hop appears in this scenario as well. After the routing table is updated, it still exists in the routing table. When the bug in the next-hop calculation is fixed, this test will probably pass as well.

6.7.1. Earlier Results

As mentioned before, the test suite has two main purposes. It can be used for conformance testing and for regression testing as well. Therefore, the tests are executed, everytimes there is a new version of OSPF. Besides of the results presented above, the test suite helped uncovering a number of bugs during the development of OSPF. Because the test suite covers the most parts of the RFC, performing the tests guarantees, that wide parts of the actual code are exected at some time. Especially the handling of exceptions or code dealing with special configurations can be tested in this way. Indeed, the execution of the test cases helped finding some unhandled exceptions. Mostly these were "standard" bugs, like unhandled `NullPointerExceptions`, `ConcurrentModification-Exceptions` or casts to wrong classes, which could be fixed easily. This helped to make the code more stable. Surprisingly only a small number of errors was related to misunderstanding the RFC.

One error for example, caused most of the Flooding tests to fail. When an LSA has to be flooded, it is set on the retransmission list. Then the a flood timer is started, which sends all LSAs from this list, when it fires. Then this timer must be set again, since SSFNet timers are single shot timers. Again, a "standard" bug caused the whole flooding procedure to fail. When the retransmission list is empty, the timer is not restarted. But the timer object exists further on. When the timer had now to be restarted, it was checked for a `null` value, instead of testing, whether the timer is cancelled. Because of this, the flooding timer was never restarted and no LSAs are flooded at all. After this bug was fixed, most flooding tests were passed.

These experiences show, how useful extensive testing is for the development process. Using the appropriate tests helps finding errors early. Detailed test cases help restricting the possible sources of errors to well defined regions of the source code.

Table 6.3.: Test results for OSPF version 0.1.4

Test No.	**Verdict**	**Problems / Remarks**
1.1	FAIL	multicast support
1.2	INCONCLUSIVE	virtual links
1.3	INCONCLUSIVE	multi-access networks
1.4	INCONCLUSIVE	multi-access networks
1.5	INCONCLUSIVE	multi-access networks
1.6	INCONCLUSIVE	multi-access networks
1.7	INCONCLUSIVE	multi-access networks
1.8	INCONCLUSIVE	multi-access networks
1.9	INCONCLUSIVE	multi-access networks
1.10	PASS	
1.11	PASS	automatic address assignment
1.12	PASS	
2.1	INCONCLUSIVE	multiaccess networks
2.2	INCONCLUSIVE	virtual links
2.3	FAIL	MTU not handled
2.4	PASS	
2.5	PASS	
2.6	PASS	
2.7	PASS	
2.8	PASS	
2.9	INCONCLUSIVE	AS-external LSAs
2.10	PASS	
2.11	INCONCLUSIVE	AS-external LSAs
2.12	INCONCLUSIVE	multiaccess networks
2.13	INCONCLUSIVE	multiaccess networks
2.14	INCONCLUSIVE	multiaccess networks
2.15	FAIL	multicast
2.16	PASS	
2.17	PASS	
2.18	FAIL	BadLsReq handling
2.19	PASS	
2.20	PASS	
2.21	FAIL	LSAs are not retransmitted
3.1	INCONCLUSIVE	multiaccess networks
3.2	INCONCLUSIVE	multiaccess networks
3.3	INCONCLUSIVE	multiaccess networks
3.4	INCONCLUSIVE	network LSA
3.5	INCONCLUSIVE	network LSA
3.6	INCONCLUSIVE	summary LSA
3.7	INCONCLUSIVE	summary LSA
3.8	INCONCLUSIVE	summary LSA

Test No.	Verdict	Problems / Remarks
3.9	INCONCLUSIVE	summary LSA
3.10	INCONCLUSIVE	summary LSA
3.11	INCONCLUSIVE	summary LSA
3.12	INCONCLUSIVE	summary LSA
3.13	INCONCLUSIVE	summary LSA
3.14	INCONCLUSIVE	summary LSA
3.15	INCONCLUSIVE	virtual links
3.16	INCONCLUSIVE	AS external LSA
3.17	INCONCLUSIVE	AS external LSA
3.18	INCONCLUSIVE	AS external LSA
3.19	INCONCLUSIVE	summary LSA
3.20	INCONCLUSIVE	summary LSA
3.21	INCONCLUSIVE	AS external LSA
3.22	FAIL	no max-aging
4.1	INCONCLUSIVE	inter-area routing
4.2	INCONCLUSIVE	inter-area routing and virtual links
4.3	INCONCLUSIVE	AS external routes
4.4	INCONCLUSIVE	AS external routes
4.5	INCONCLUSIVE	AS external routes
4.6	INCONCLUSIVE	AS external routes
4.7	INCONCLUSIVE	AS external routes
4.8	INCONCLUSIVE	AS external routes
4.9	INCONCLUSIVE	AS external routes
5.1	PASS	implicit
5.2	PASS	implicit
5.3	INCONCLUSIVE	AS external LSA and virtual links
5.4	INCONCLUSIVE	IP precedence
5.5	INCONCLUSIVE	virtual links
5.6	INCONCLUSIVE	RFC1583
5.7	INCONCLUSIVE	authentication
5.8	INCONCLUSIVE	checksum
5.9	PASS	implicit
5.10	PASS	implicit
5.11	PASS	implicit
5.12	PASS	implicit
5.13	INCONCLUSIVE	TOS
5.14	PASS	
6.1	PASS	
6.2	FAIL	wrong next-hop calculation
6.3	FAIL	no packet forwarding
6.4	FAIL	same error as 6.2

7. Conclusions

7.1. Summary

The goal of this thesis was to analyze how routing protocols can be tested. An additional goal was to provide the SSFnet simulation framework with a test environment for its OSPF implementation. Only when a simulation behaves exactly as the real world does, it is possible to gain significant information about the reality from a simulation. Therefore it is indispensable, that the implementations of the protocols used in the simulation are compliant to their specifications.

First, an overview of the SSFnet framework was given. It was shown, how SSFnet works and how topologies for the simulations can be configured. To provide a test environment for SSFNet, the different concepts and methods for conformance testing had to be analyzed. Formal and informal approaches can be used for the derivation of test cases. Formal methods in conformance testing have the advantage, that they prove an implementation to be conformant to its standard. But these formal methods are often very difficult to apply, especially when there is no formal specification of the protocol. Informal methods do not assure, that a protocol which passes the tests is really conformant to the specification. Anyway, they provide a certain degree of certainty, that the implementation behaves properly.

Implementing a test suite for a routing protocol requires an understanding of the way, these protocols work. All routing protocols have some common functionalities. According to this common structure, general test purposes for conformance testing of arbitrary routing protocols were proposed. After that, the OSPF protocol was analyzed with respect to testing. There are many commercial testing products on the market. Some of them can also be used for OSPF conformance testing. However, they often do not use formal methods, so the coverage of the tests cannot be proven formally. Another drawback of these commercial products is their price.

To implement a test suite for OSPF, a conceptual framework for testing within the SSFNet framework was proposed. This framework is consitent with the methods for OSI conformance testing. After that, a test suite had to be chosen. The commercial products were not considered here, since they cannot be integrated into the SSFNet framework easily. The decision was made to use the OSPF test suite from the University of New Hampshire's InterOperabiliy Lab. This test suite can be tranferred easily into the SSFnet framework. In fact, no formal methods were used to build this test suite, but since many vendors of networking products use it to test their devices, it provides a good degree of

certainty that the OSPF implementation in SSFnet behaves as desired, when it passes the tests.

Most parts of the OSPF specification deal with the dynamic behavior in networks, such as link failures and topology changes. At the moment, the support for such behavior in SSFnet is very poor. For example, there are no links, that can be brought up or taken down at certain times. Network interfaces cannot be stopped and started and routers cannot be restarted. To model these behaviors several classes were implemented to add this functionality. Additional classes were implemented to simulate certain test behaviors which could not be modeled with the standard configuration features provided by SSFnet.

Running the tests finally showed, that it is always a good idea to perform extensive testing for a complex implementation like this. Although most parts of the OSPF standard were implemented correctly, a variety of errors could be found and eliminated from the implementation. Since the test suite covers most of the possible behaviors from the protocol standard, wide parts of the source code have to be executed for the execution of the test suite. That assures, that many exceptions can be found and caught in the implementation. Thus, the test suite can be used for three types of testing. During the development of the OSPF implementation, it can be used for debugging purposes. After that, it can be used to show, that the implementation complies to its specification. The last application of the test suite is regression testing, which is needed everytimes, the underlying SSFNet classes are changed. However, the test suite does not assure, that the resulting code is bug-free after passing the tests. More extensive testing (e.g. stress tests, etc.) is needed here.

7.2. Open Problems and Outlook

The implementation of the test suite showed, that testing is an important part of the implementation of a complex protocol. A drawback of current testing of Internet protocols is that there is no standardized method for conformance testing. Indeed, the OSI conformance testing framework is applicable to Internet standards as well. A problem is the derivation of test cases from the protocol standard, which is not standardized within the OSI framework. It would be desirable to use some formal method for test case selection, because formal methods allow to prove the coverage and the correctness of a test suite. But therefore, the protocol must be specified using some formal description technique, which is not the case for common Internet protocols. A solution for this problem would be to provide every protocol standard with a formal specification or at least with a standardized conformance test suite.

From the beginning it was clear, that the current OSPF implementation does not support all features from the OSPF standard as specified in RFC 2328. Some features, like authentication, checksums and virtual links are not important at all to get meaningful results out of a simulation. For the sake of completeness, these features can be added later. More crucial features, like multiaccess networks, inter-area routing and the import and use of externally derived routing information (e.g. OSPF/BGP interaction) will be implemented

soon. Due to the lack of multiaccess networks, some test cases had to modified. These test cases have again to be changed, when the according features are added to OSPF.

A. SSF API

A.1. Entity

```
public interface Entity {
    public ltime_t now();
    public void startAll( ltime_t t1 );
    public void startAll( ltime_t t0, ltime_t t1 );
    public ltime_t pauseAll();
    public void resumeAll();
    public void joinAll();
    public ltime_t alignto( Entity s );
    public Object alignment();
    public java.util.Vector coalignedEntities();
    public void init();
    public java.util.Vector processes();
    public java.util.Vector inChanels();
    public java.util.Vector outChannels();
}
```

A.2. Event

```
public interface Event {
    public Event save();
    public void release();
    public boolean aliased();
}
```

A.3. inChannel

```
public interface inChannel {
    public Entity owner();
    public Event[] activeEvents();
    public outChannel[] mappedto();
}
```

A.4. outChannel

```
public interface outChannel {
```

```
    public Entity owner();
    public inChannel[] mapedto();
    public void write( Event evt, ltime_t delay );
    public void write( Event evt );
    public ltime_t mapto( inChannel tgt );
    public ltime_t mapto( inChannel tgt, time_t mapping_delay );
    public ltime_t mapto( inCHannel tgt );
}
```

A.5. process

```
public interface process {
    public Entity owner();
    public void action();
    public void init();
    public void waitOn( inChannel[] waitchannels );
    public void waitOn( inChannel waitchannel );
    public void waitForever();
    public void waitFor( ltime_t waitinterval );
    public boolean waitOnFor( inChannel[] waitchannels,
                              ltime_t timeout );
    public boolean isSimple();
}
```

B. SSF.OS API

B.1. ProtocolSession

```
public abstract class ProtocolSession
implements Configurable {
  public void init();
  public abstract boolean push(ProtocolMessage message,
                               ProtocolSession fromSession);
  public void config(Configuration cfg);
  public String version();
  public ProtocolGraph inGraph();
  public void setGraph(ProtocolGraph G);
  public void open(ProtocolSession S, Object request);
  public void opened(ProtocolSession S);
  public void close(ProtocolSession S);
  public void closed(ProtocolSession S);
}
```

B.2. ProtocolGraph

```
public class ProtocolGraph extends Entity
implements Configurable {
  public ProtocolGraph();
  public void config(Configuration cfg);
  public void init();
  public ProtocolSession SessionForName(String protocol_name);
}
```

B.3. ProtocolMessage

```
public class ProtocolMessage {
  public float size();
  public String version();
  public static ProtocolMessage fromVersion(String V);
  public ProtocolMessage();
  public void dropPayload();
  public void carryPayload(ProtocolMessage payload);
  public ProtocolMessage payload();
  public ProtocolMessage previous();
```

```
  public int bytecount();
  public void tobytes(byte[] buf, int offset);
  public void frombytes(byte[] buf, int offset);
}
```

B.4. PacketEvent

```
public class PacketEvent extends Event {
  public ProtocolMessage getHeaders();
  public PacketEvent();
  public PacketEvent(ProtocolMessage msg);
  public PacketEvent(Class pktclass);
}
```

B.5. Timer

```
public abstract class Timer extends Entity {
  public boolean isCancelled();
  public abstract void callback();
  public void set();
  public void set(long dt);
  public Timer(ProtocolGraph e, long dt);
  public final void cancel();
}
```

B.6. Continuation

```
public interface Continuation {
  public void success();
  public void failure(int errno);
}
```

C. OSPF Packets

C.1. OSPF Packet Header

8 bit	*8 bit*	*8 bit*	*8 bit*
Version	Type	Length	
Router ID			
Area ID			
Checksum		Authentication Type	
Authentification Data			
Authentification Data			
Packet Data			

Figure C.1.: The OSPF Packet Header

C.2. Hello Packets

8 bit	*8 bit*	*8 bit*	*8 bit*
Version	Type = 1	Length	
Router ID			
Area ID			
Checksum		Authentication Type	
Authentification Data			
Authentification Data			
Network mask			
Hello Interval		Options	Priority
Router Dead Interval			
Designated Router			
Backup Designated Router			
Neighbor 1 Router ID			
⋮			
Neighbor n Router ID			

Figure C.2.: The Hello Packets

C.3. Database Description Packets

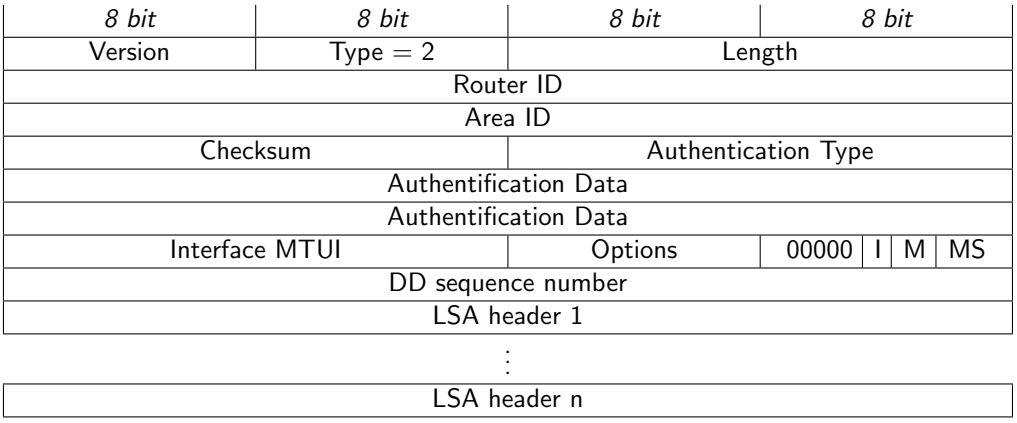

Figure C.3.: The Database Description Packet

C.4. Link State Request Packets

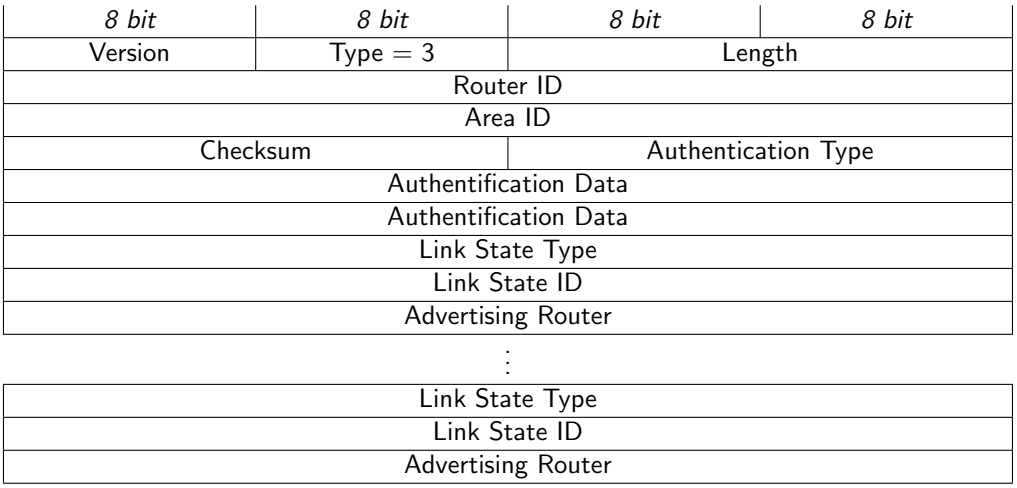

Figure C.4.: The LSRequest Packet

C.5. Link State Update Packets

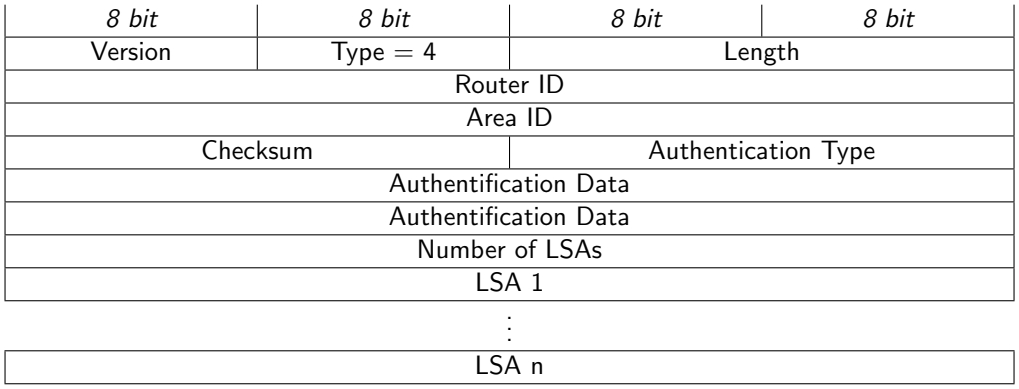

Figure C.5.: The LSUpdate Packet

C.6. Link State Acknowledgement Packets

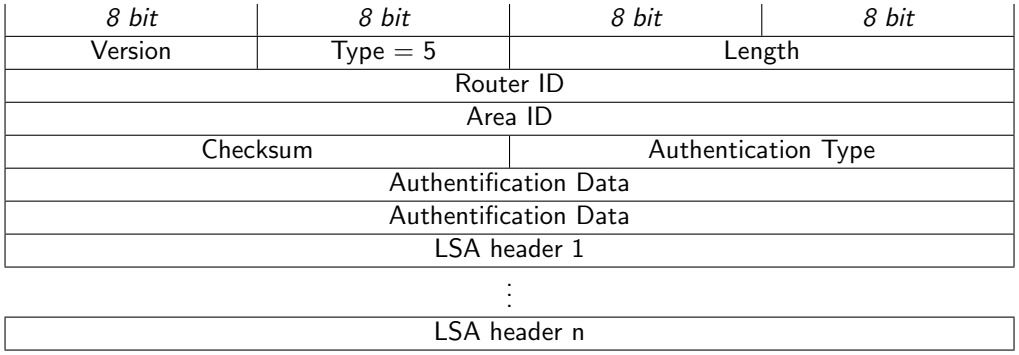

Figure C.6.: The LSAcknowledgement Packet

C.7. Link State Advertisements

C.7.1. LSA Header

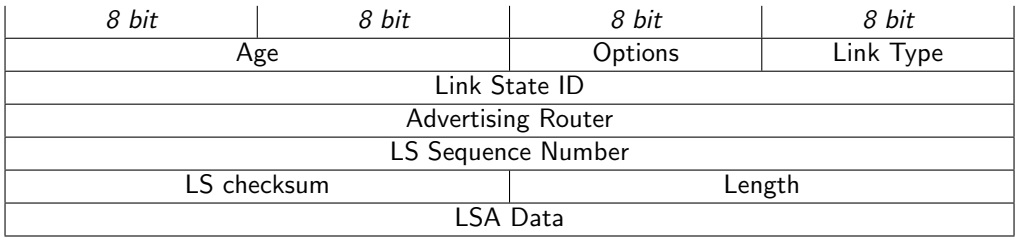

Figure C.7.: The LSA Header

C.7.2. Router LSA

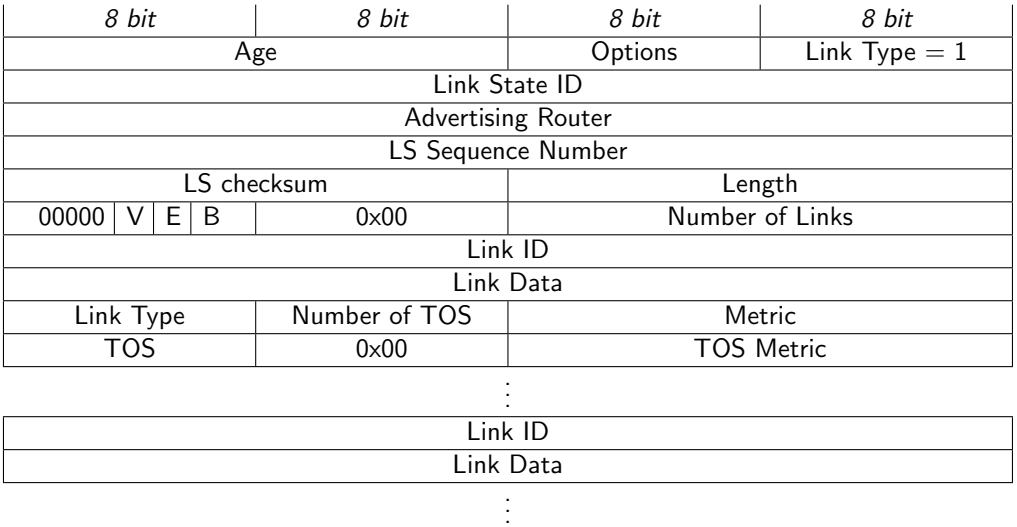

Figure C.8.: The Router LSA

C.7.3. Network LSA

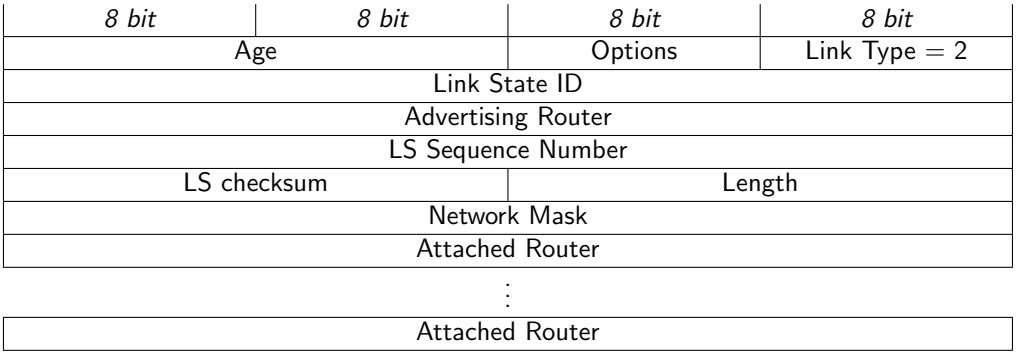

Figure C.9.: The Network LSA

C.7.4. Network Summary LSA

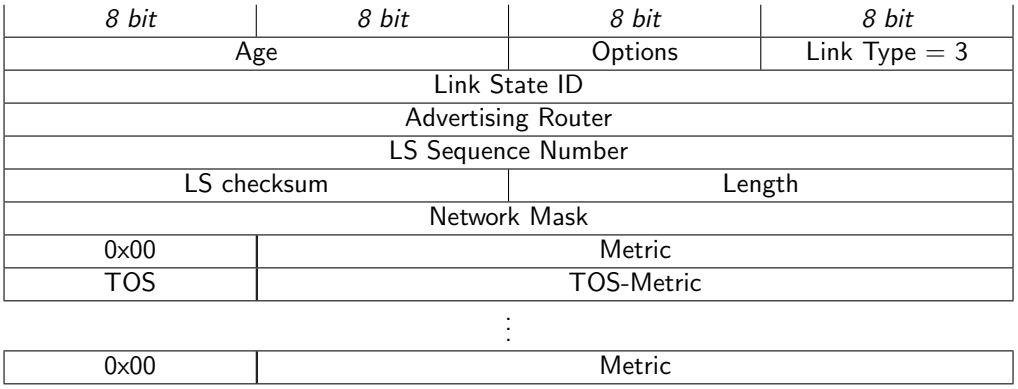

Figure C.10.: The Network Summary LSA

C.7.5. ASBR Summary LSA

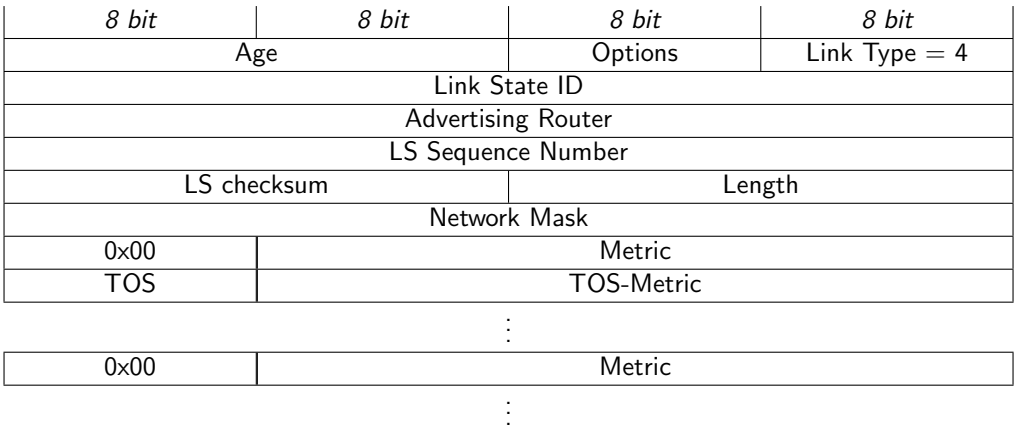

Figure C.11.: The ASBR Summary LSA

C.7.6. AS External LSA

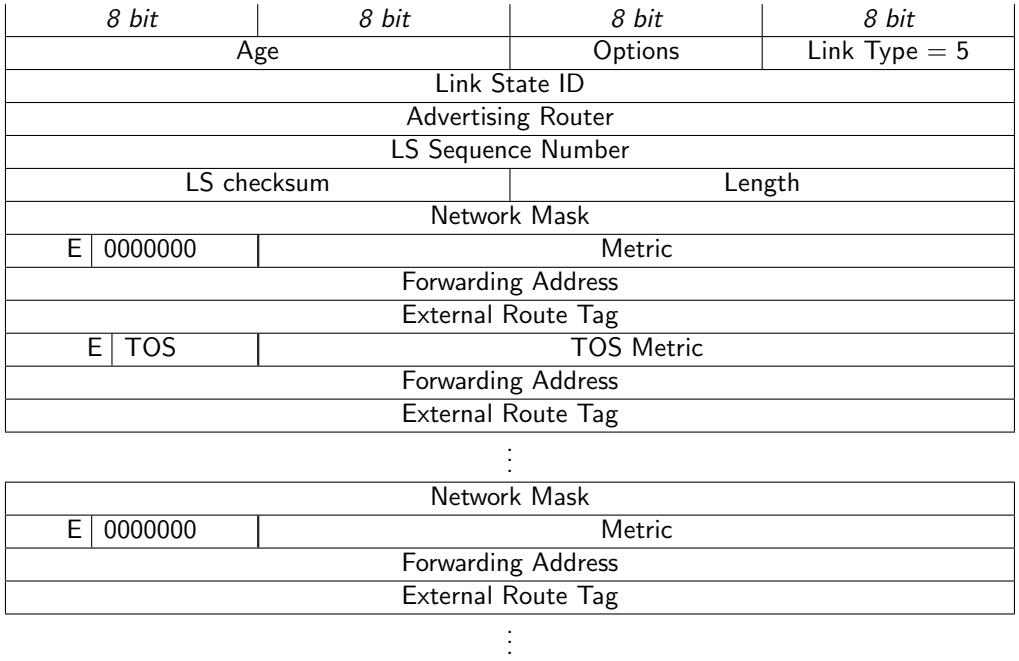

Figure C.12.: The AS External LSA

D. Test Cases

This appendix contains a list of all test cases from the IOL OSPF test suite [Int00], together with their test purpose description.

D.1. Hello Protocol Tests

Test 1.1 – hello_addressing_and_timing
Verify that the Hello packets are sent every HelloInterval seconds to the IP multicast address AllSPFRouters on broadcast and point-to-point networks.

Test 1.2 – hello_virtual
Verify that on virtual links Hello packets are sent as unicasts every HelloInterval seconds.

Test 1.3 – hello_waiting_state
Verify that a router does not elect DR or BDR until it transitions out of Waiting state.

Test 1.4 – event_backupseen
Verify that event BackupSeen occurs properly and brings an interface out of state Waiting.

Test 1.5 – no_waiting
Verify that if a router has priority 0 on an interface, the interface state machine does not go through state Waiting but goes directly to DR Other.

Test 1.6 – accept_existing_dr
Verify that when a router's interface to a network first becomes functional, if there is already an existing DR, it accepts that DR regardless of its own priority.

Test 1.7 – dr_collision
Verify that if two or more routers have declared themselves DR, the one with the highest priority is chosen to be DR. In case of a tie, the one having the highest Router ID is chosen.

Test 1.8 – bdr_become_dr
Verify that the BDR becomes DR when the previous DR fails.

Test 1.9 – dr_other_become_bdr

Verify that when the DR fails, the DR Other with the highest priority becomes BDR, and synchronizes its database with all other routers on the network except the new DR.

Test 1.10 – hello_mismatch
Verify that any mismatch between the Hello packet values Area ID, Network Mask, HelloInterval, RouterDeadInterval, and the configuration of the receiving interface cause the packet to be dropped as long as the interface is not part of a point-to-point network or a virtual link.

Test 1.11 – remote_hello
Verify that if an incoming OSPF packet is not from a local network then it is discarded.

Test 1.12 – hello_e_bit
Verify that the E bit of the Options field in a Hello packet is set if and only if the attached area is not a stub area. If two routers on a network do not agree on the E bit, they will not become neighbors.

D.2. Flooding and Adjacency Tests

Test 2.1 – multi_access_adjacencies
Verify that on a multi-access network, the DR and BDR become adjacent with all other routers, while a DR Other only becomes adjacent with the DR and BDR.

Test 2.2 – database_description_mtu_set
Verify that a router properly sets the MTU for its interface to a network in its Database Description packets.

Test 2.3 – database_description_mtu_match
Verify that a router properly identifies the MTU for its interface to a network in its Database Description packets, and any incoming DD packet with an MTU set higher than this value will be dropped.

Test 2.4 – master_negotiation
Verify that the Master/Slave negotiation is done correctly.

Test 2.5 – self_orig_lsa_rcpt
Verify that a router advances its LS sequence numbers when it finds that there are old LSAs originated by itself in another router's database.

Test 2.6 – old_lsa_rcpt
Verify that a router discards an LSA that is older than the database copy if it supports only RFC 1583. If the router supports RFC2328, the RUT should send its current database copy of the LSA unicast back to a neighbor from whom it receives an LSA that is older than the database copy.

Test 2.7 – nbr_state_too_low
Verify that a router discards an LSA or LS Request received from a neighbor in a lesser state than Exchange.

Test 2.8 – dd_retransmit
Verify that a router properly retransmits Database Description packets.

Test 2.9 – event_seq_number_mismatch
Verify that a router transitions to state ExStart when Event SeqNumberMismatch occurs.

Test 2.10 – flooding
Verify that a router properly floods non-AS-external-LSAs throughout the area but not outside of it.

Test 2.11 – flooding_ase
Verify that a router properly floods AS-external-LSAs throughout the OSPF Autonomous System.

Test 2.12 – ls_ack
Verify that a router properly floods or acknowledges an incoming LSA.

Test 2.13 – lsa_retransmission
Verify that a router properly places all routers that it is adjacent with on its retransmission list when appropriate.

Test 2.14 – lsa_flooding_guarantee
Verify that a router properly places all routers that it is adjacent with on its retransmission list when appropriate.

Test 2.15 – lsa_multicast
Verify that a router sends its LS Update packets to the correct multicast address depending on the state of its interface.

Test 2.16 – lsa_retransmit_unicast
Verify that a router sends all retransmitted LSAs in unicast Link State Update packets.

Test 2.17 – lsa_request_retransmit
Verify that a router retransmits an unsatisfied LS Request every RxmtInterval seconds, and that an LSA is removed from its LS Request List upon reception of a valid Link State Update containing that LSA.

Test 2.18 – event_bad_ls_request
Verify that a router transitions to state ExStart when event BadLSReq occurs.

Test 2.19 – lsa_maxage_flood
Verify that a router properly floods an LSA when its age reaches MaxAge.

Test 2.20 – lsa_refresh
Verify that a router properly sends a new instance of a self-originated LSA when its age reaches LSRefreshTime in its link state database.

Test 2.21 – remove_lsa_from_retransmission_list
Verify that a router removes an LSA from its link state retransmission list when that LSA has been removed from its link state database.

D.3. Link State Advertisement Tests

Test 3.1 – router_lsa_transit
Verify that a router sends a new router-LSA when an attached network changes from a stub network to a transit network.

Test 3.2 – router_lsa_dr_change
Verify that when a network's DR changes, a router sends a new router-LSA with a type 2 link, containing the IP address of the new DR in the Link ID.

Test 3.3 – router_lsa_stub
Verify that a router properly identifies a directly connected stub network with a type 3 link in its router-LSA.

Test 3.4 – net_lsa_dr_change
Verify that a router originates a network-LSA for a network on which it is DR and has at least one adjacent neighbor.

Test 3.5 – net_lsa_attached_router
Verify that when a router originates a network-LSA, it lists all of those routers with which it is fully adjacent in the LSA.

Test 3.6 – summary_asbr_lsa_intra_area
Verify that an Area Border Router properly originates a summary-ASBR-LSA when it has an intra-area route to an ASBR.

Test 3.7 – summary_net_lsa_intra_area
Verify that an Area Border Router properly originates a summary-network-LSA for a network for which it has an intra-area route.

Test 3.8 – summary_asbr_lsa_inter_area
Verify that an Area Border Router properly originates a summary-ASBR-LSA when it has an inter-area route to an ASBR.

Test 3.9 – summary_net_lsa_inter_area
Verify that an Area Border Router properly sends summary-network-LSAs for those networks reachable by inter-area routes.

Test 3.10 – intra_area_become_inter_area
Verify that an Area Border Router properly sends a summary-network-LSA for a network for which it previously had an intra-area route but now only has an inter-area route (due to an interface going down).

Test 3.11 – summary_area_range_rfc1583
Verify that an Area Border Router properly uses a configured address range, as per RFC 1583.

Test 3.12 – summary_area_range_new
Verify that an Area Border Router properly uses a configured address range, as per RFC 2328.

Test 3.13 – summary_area_range_common
Verify that an Area Border Router properly flushes any advertisements it originated for a configured address range when all of the component networks become unreachable.

Test 3.14 – summary_area_range_no_transit
Verify that an Area Border Router does not summarize backbone networks to transit areas.

Test 3.15 – virtual_link
Verify that a router correctly forms and maintains a virtual link, and properly advertises changes in the virtual link through its router-LSAs.

Test 3.16 – ase_static
Verify that an ASBR properly originates AS-external-LSAs for static routes with the configured type, cost and forwarding address.

Test 3.17 – ase_rip
Verify that an Autonomous System Boundary Router (ASBR) properly advertises externally learned destinations.

Test 3.18 – remove_redundant_ase
Verify that an ASBR flushes its own AS-external-LSA when another ASBR with higher router ID originates a functionally equivalent AS-external-LSA.

Test 3.19 – default_summary_origination
Verify that an ABR connected to a stub area properly originates a default summary-LSA into the stub area with StubDefaultCost when configured to do so.

Test 3.20 – default_summary_use
A router internal to a stub area should correctly use a default summary-LSA.

Test 3.21 – host_bits_ase
Verify that a router properly handles and sets Host Bits in AS-external-LSAs.

Test 3.22 – ls_sequence_number_wrap
Verify that when a router must increment the LS sequence number of a self-originated LSA past 0x7FFFFFFF, it flushes the current instance of the LSA from the routing domain before originating a new instance with sequence number 0x80000001.

D.4. Routing Calculation Tests

Test 4.1 – prefer_intra_area_route
Verify that a router prefers intra-area OSPF routes to inter-area OSPF routes.

Test 4.2 – inter_area_through_transit
Verify that a router properly calculates inter-area routes when it is an ABR attached to a transit area.

Test 4.3 – ase_forwarding_address
Verify that a properly uses the ForwardingAddress field in AS-external-LSAs.

Test 4.4 – asbr_multiple_intra_area
Verify that when multiple intra-area paths to an ASBR are available, a router chooses the correct path.

Test 4.5 – prefer_internal
Verify that a router chooses the correct type of route when OSPF internal and external routes exist to a network.

Test 4.6 – ase_type_1_and_2
Verify that when choosing between multiple ASBRs advertising routes to the same destination, the router prefers type 1 routes over type 2. If only type 2 costs are present, a router always chooses the path to the ASBR advertising the lowest type 2 metric.

Test 4.7 – multiple_asbr_intra_area_1583_enabled
Verify that a router properly chooses between multiple ASBRs when RFC 1583 compatibility is enabled.

Test 4.8 – multiple_asbr_intra_1583_disabled
Verify that a router properly chooses between multiple ASBRs reachable via intra-area non-backbone routes when RFC 1583 compatibility is disabled.

Test 4.9 – multiple_asbr_inter
Verify that a router properly chooses between multiple ASBRs reachable through inter-area or intra-area backbone routes when RFC 1583 compatibility is enabled or disabled.

D.5. Configuration and Formatting Tests

Test 5.1 – area_parameters
Verify that the area parameters are configurable.

Test 5.2 – interface_parameters
Verify that the interface parameters are configurable.

Test 5.3 – router_lsa_bits
Verify that a router properly sets the E, B and V bits in its router-LSAs.

Test 5.4 – precedence_tos_ip_header
Verify that a router properly sets the TOS and Precedence fields in the IP header.

Test 5.5 – no_virual_link_in_stub
Verify that a router does not allow a virtual link to be configured in a stub area.

Test 5.6 – authentication_rfc1583
Verify that authentication type is configurable on a per-area basis, and additional authentication data is configurable on a per-interface basis.

Test 5.7 – authentication_rfc2328
Verify that authentication type and additional authentication data is configurable on a per-interface basis.

Test 5.8 – checksum
Verify the handling of the Checksum field.

Test 5.9 – #advertisements_field
Verify the handling of the #Advertisements field in LS Update packets.

Test 5.10 – packet_length_field
Verify the handling of the PacketLength field in OSPF packets.

Test 5.11 – lsa_header_length
Verify the handling of the Length field in the LSA header.

Test 5.12 – router_lsa_#links_field
Verify the handling of the #Links field in router-LSAs.

Test 5.13 – router_lsa_#tos_field
Verify the handling of the #TOS field in router-LSAs.

Test 5.14 – bad_lsa_age
Verify the handling of the LSAge field in LSA packets.

Bibliography

[ADLU94] Alfred V. Aho, Anton T. Dahbura, David Lee, and M. Ümit Uyar. An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours. In *Conformance Testing Methodologies and Architectures for OSI Protocols*. IEEE Computer Society Press, 1994.

[Agi00] Agilent Technologies. *Internet Routing Protocol Testing Whitepaper*, 2000.

[Agi01a] Agilent Technologies. *Integrating Agilent's RouterTester & QARobot*, 2001.

[Agi01b] Agilent Technologies. *OSPF Conformance Test Suite – Product Data Sheet*, 2001.

[Ard97] M.A. Ardis. *Formal Methods for Telecommunication System Requirements: A survey of standardized Languages*. Bell Laboratories, 1997.

[BSS94] Ed Brinksma, Giuseppe Scollo, and Chris Steenbergen. LOTOS Specifications, their Implementations and their Tests. In *Conformance Testing Methodologies and Architectures for OSI Protocols*. IEEE Computer Society Press, 1994.

[BWC98] Jun Bi, Jianping Wu, and Xiuhuan Chen. A Concurrent TTCN based Approach to Conformance Testing of Distributed Routing Protocol OSPF v2. In *Proceedings of the International Conference on Computer Communications and Networks*, 1998.

[CFP93] A.R. Cavalli, J.-P. Favreau, and M. Phalippou. Formal Methods in Conformance Testing: Result and Perspectives. *Protocol Test Systems*, 1993.

[Cho94] Tsun S. Chow. Testing Software Design Modeled by Finite-State Machines. In *Conformance Testing Methodologies and Architectures for OSI Protocols*. IEEE Computer Society Press, 1994.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[CNO99a] J.H. Cowie, D.M. Nicol, and A.T. Ogielski. Modeling 100000 Nodes and Beyond: Self-Validating Design. In *DARPA/NIST Workshop on Validation of Large Scale Network Simulation Models*, May 1999.

[CNO99b] J.H. Cowie, D.M. Nicol, and A.T. Ogielski. Modeling the Global Internet. *Compting in Science and Engineering*, 1(1), January 1999.

[Cow99] J.H. Cowie. *Scalable Simulation Framework API Reference Manual*. S3 Consortium, March 1999.

[DCB91] L. Drayton, A. Chetwynd, and G. Blair. *An introduction to LOTOS through a worked example*. Lancaster University, 1991.

[Doy98] Jeff Doyle. *CCIE Professional Development: Routing TCP/IP*, volume 1. Cisco Press, September 1998.

[Emp00] Empirix. *ANVL User's Guide for IP Test Suites*, 2000.

[Emp01a] Empirix. *ANVL OSPF Test Suite – Product Data Sheet*, 2001.

[Emp01b] Empirix. *Automated Network Validation Library – Product Data Sheet*, 2001.

[FUA+99] M.A. Fecko, M.Ü. Uyar, P.D. Amer, A.S. Sethi, T. Dzik, R. Menell, and M. McMahon. A Success Story of Formal Description Techniques: Estelle Specification and Test Generation for MIL-STD 188-220. *Computer Communications*, 1999.

[FvBK+94] S. Fujiwara, G. v. Bochmann, F. Kehndek, M. Amalou, and A. Ghedamsi. Test Selection based on Finite State Models". In *Conformance Testing Methodologies and Architectures for OSI Protocols*. IEEE Computer Society Press, 1994.

[GHN93] Jens Grabowski, Dieter Hogrefe, and Robert Nahm. A Method for the Generation of Test Cases Based on SDL and MSCs. Technical report, Institut für Informatik, Universität Bern, 1993.

[GHNS93] Jens Grabowski, Dieter Hogrefe, Robert Nahm, and Andreas Spichiger. Relating Test Purposes to Formal Specifications: Towards a Theoretical Foundation of Practical Testing. Technical report, Institut für Informatik, Universität Bern, 1993.

[GKSH98] J. Grabowski, B. Koch, M. Schmitt, and D. Hogrefe. SDL and MSC Based Test Generation for Distributed Test Architectures. Technical report, Medical University of Lübeck, Institute for Telematics, 1998.

[GL94] Djaffar Gueraichi and Luigi Logrippo. Derivation of Test Cases for LAP-B from a LOTOS Specification. In *Conformance Testing Methodologies and Architectures for OSI Protocols*. IEEE Computer Society Press, 1994.

[GNSH93] Jens Grabowski, Robert Nahm, Andreas Spichinger, and Dieter Hogrefe. *Die SAMSTAG Methode und ihre Rolle im OSI Konformitätstesten.* Institut für Informatik, Universität Bern, 1993.

[Gön94] Güney Gönenç. A Method for the Design of Fault Detection Experiments. In *Conformance Testing Methodologies and Architectures for OSI Protocols.* IEEE Computer Society Press, 1994.

[GW99] J. Grabowski and T. Walter. Towards an Integrated Test Methodology for Advanced Distributed Systems. In *Proceedings of the 16th International Conference and Exposition on Testing Computer Software*, 1999.

[Hen94] F. C. Hennie. Fault Detecting Experiments for Sequential Circuits. In *Conformance Testing Methodologies and Architectures for OSI Protocols.* IEEE Computer Society Press, 1994.

[HP92] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *ACM Transactions on Computer Systems*, 10(2), May 1992.

[Int83] International Organization for Standardization. *ISO 9646: Information Technology – Open System Interconnection – Conformance Testing Methodology and Framework*, 1983.

[Int94] International Organization for Standardization. *ISO 7498: Information Processing Systems – Open Systems Interconnection – Basic Reference Model*, 1994.

[Int99a] International Telecommunication Union. *Z.100: Formal Description Techniques (FDT) – Specification and Description Language (SDL)*, 1999.

[Int99b] International Telecommunication Union. *Z.120: Formal Description Techniques (FDT) – Message Sequence Chart (MSC)*, 1999.

[Int00] Intenet Protocol Consortium, InterOperability Laboratory, University of New Hampshire. *IP Consortium Test Suite: Open Shortest Path First*, April 2000.

[IP 01] IP Infusion Inc. *ZebOS Advanced Routing Suite: Testing & Interoperability Whitepaper*, 2001.

[KR00] James F. Kurose and Keith W. Ross. *Computer Networking: A top down approach featuring the Internet.* Addison-Wesley, 2000.

[LDvB+93] Gang Luo, Rachida Dssouli, Gregor v. Bochmann, Pallapa Venkataram, and Abderrazak Ghedamsi. *Generating Synchronizable Test Sequences Based on Finite State Machines with Distributed Ports*, 1993.

[Lin94] Richard J. Linn. Conformance Testing for OSI Protocols. In *Conformance Testing Methodologies and Architectures for OSI Protocols*. IEEE Computer Society Press, 1994.

[LN01] J. Liu and D.M. Nicol. *Dartmouth Scalable Simulation Framework Version 3.1 - User's Manual*. Dartmouth College, Department of Computer Science, May 2001.

[LU94a] Richard J. Linn and M. Ümit Uyar, editors. *Conformance Testing Methodologies and Architectures for OSI Protocols*. IEEE Computer Society Press, 1994.

[LU94b] Richard J. Linn and M. Ümit Uyar, editors. *Conformance Testing Methodologies and Architectures for OSI Protocols*, pages 277–321. IEEE Computer Society Press, 1994.

[McK84] Alex McKenzie. *RFC 905: ISO Transport Protocol Specification ISO DP 8073*, April 1984.

[MG99] Mazen Malek and Roland Gecse. Testing Internet Applications - Terminology and Applicability. *Acta Cybernetica*, 14(2), 1999.

[Moy94] John T. Moy. *RFC 1583: OSPF Version 2*, March 1994.

[Moy97] John T. Moy. *OSPF: Anatomy of an Internet Routing Protocol*. Addison-Wesley, 1997.

[Moy98] John T. Moy. *RFC 2328: OSPF Version 2*, April 1998.

[OP91] S. W. O'Malley and L. L. Peterson. A Dynamic Network Architecture. *IEEE Transactions on Software Engineering*, 17(1), January 1991.

[Pos80] Jon Postel. *RFC 768: User Datagram Protocol*, August 1980.

[Pos81] Jon Postel. *RFC 793: Transmission Control Protocol*, September 1981.

[Ray94] D. Rayner. OSI Conformance Testing. In *Conformance Testing Methodologies and Architectures for OSI Protocols*. IEEE Computer Society Press, 1994.

[RL95] Yakov Rekhter and Tony Li. *RFC 1771: A Border Gateway Protocol 4 (BGP-4)*, March 1995.

[SD94] Krishnan Sabnani and Anton Dahbura. A Protocol Test Generation Procedure. In *Conformance Testing Methodologies and Architectures for OSI Protocols*. IEEE Computer Society Press, 1994.

[SG00] I. Schieferdecker and J. Grabowski. *Conformance Testing with TTCN*. GMD FOKUS, Institute for Telecooperation Technology, 2000.

[Ste99] John W. Stewart. *BGP4: Inter Domain Routing in the Internet*. Addison-Wesley, 1999.

[SvB94] B. Sarikaya and G. v. Bochmann. Some Experience with Test Sequence Generation for Protocols. In *Conformance Testing Methodologies and Architectures for OSI Protocols*. IEEE Computer Society Press, 1994.

[SvBC94] Behçet Sarikaya, Gregor von Bochmann, and Eduard Cerny. A Test Design Methodology for Protocol Testing. In *Conformance Testing Methodologies and Architectures for OSI Protocols*. IEEE Computer Society Press, 1994.

[Tan97] Andrew S. Tanenbaum. *Computernetzwerke*. Prentice Hall, 1997.

[UD94] M. Ümit Uyar and Anton T. Dahbura. Optimal Test Sequence Generation for Protocols: The Chinese Postman Algorithm applied to Q.931. In *Conformance Testing Methodologies and Architectures for OSI Protocols*. IEEE Computer Society Press, 1994.

[Ura94] Hasan Ural. A Test Derivation Method for Protocol Conformance Testing. In *Conformance Testing Methodologies and Architectures for OSI Protocols*. IEEE Computer Society Press, 1994.