

OVOPS++ manual

Version 1.0 (8th November 1999)

Olli Suihkonen

TABLE OF CONTENTS

ABBREVIATIONS	2
1 INTRODUCTION.....	3
2 ARCHITECTURE.....	3
2.1 Conduits	4
2.2 Using proxies	4
2.3 Information chunks	5
2.4 Relations between classes	6
3 REFERENCE GUIDE.....	8
3.1 pfAccessor.....	8
3.2 pfAdapter.....	8
3.3 pfConduit.....	9
3.4 pfDebug.....	10
3.5 pfException.....	10
3.6 pfFactory.....	11
3.7 pfFrame.....	11
3.8 pfIE	12
3.9 pfIEcontainer.....	12
3.10 pfMessenger	12
3.11 pfMux.....	13
3.12 pfProtocol.....	13
3.13 pfState.....	13
3.14 pfSystem.....	15
3.15 pfTimer	15
3.16 pfTimerMessenger.....	15
3.17 pfTimers.....	16
3.18 pfTransporter	16
4 HOW PF AND SF WORK TOGETHER.....	18
4.1 Synchronous/asynchronous messages	18
4.2 Synchronous/asynchronous conduits	18
4.3 Timer function.....	19
4.4 Function calls between conduits	19
5 USER'S GUIDE.....	21
5.1 Some things to remember.....	21
5.2 Main program	21
5.3 Interface classes and FSM.....	21
6 EXAMPLE PROTOCOL: ISUP	23
7 OVOPS++ EXERCISE.....	24
7.1 General description.....	24
7.2 Used PF classes.....	24
7.3 Implemented classes	25
7.4 Finite state machines.....	25
7.5 Class diagrams.....	27
REFERENCES.....	29
APPENDIX 1	

ABBREVIATIONS

API	Application Programming Interface
ATM	Asynchronous Transfer Mode
CC	Call Control
CIC	Circuit Identification Code
FSM	Finite State Machine
IE	Information Element
IP	Internet Protocol
ISDN	Integrated Services Digital Network
ISUP	ISDN User Part
MTP-3	Message Transfer Part 3
NI	Network Interface
ORB	Object Request Broker
OVOPS	Object Virtual OPERations System
OVOPS++	Object Virtual OPERations System++
PF	Protocol Framework
PDU	Protocol Data Unit
SCOMS	Software CONfigurable Multidiscipline Switch
SAAL	Signalling ATM Adaptation Layer
SF	Scheduler Framework
SI	Service Indicator
SS7	Signalling System #7
STL	Standard Template Library
TOVE	Transparent Object-oriented Virtual Exchange

1 INTRODUCTION

Object Virtual Operations System++ (OVOPS++) is an object-oriented protocol framework for data communication software implementation. It has been developed mainly in Helsinki University of Technology's Telecommunication Software and Multimedia Laboratory research projects called Transparent Object-oriented Virtual Exchange (TOVE) and Software CONfigurable Multidiscipline Switch (SCOMS). OVOPS++ is based on integration of Conduits+ -framework and OVOPS. Ascom Tech AG, University of Illinois and GLUE Software Engineering developed Conduits+. It is a component-based model of a framework that uses object-oriented design patterns. OVOPS was developed by Telecom Finland and Lappeenranta University of Technology, and it solves general problems appearing in protocol implementation, providing classes for scheduling, timers, messages etc. [MPR+96] The OVOPS++ software has been made with C++ language, and it currently operates in UNIX environment.

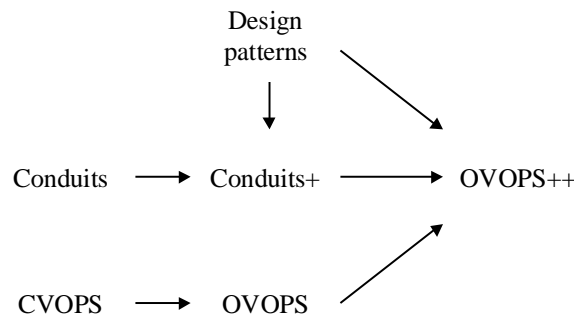


Figure 1. OVOPS++ evolution.

This document is the OVOPS++ manual. The code examples are, whenever possible, taken from the SCOMS software. Because the design patterns are an important part of the framework, their usage is explained. ISDN User Part (ISUP) protocol is used as an example of implementing a protocol with OVOPS++, as the author has implemented it. It should be noted that the examples are only single observations of OVOPS++, and the framework allows comprehensive methods for implementation.

2 ARCHITECTURE

OVOPS++ provides base classes that are used to derive protocol specific classes. It includes two different modules, scheduler framework (SF) and protocol framework (PF), which interact with each other. SF is responsible for scheduling. Its usage is only visible inside the base classes of PF, where the PF components ask for processing time. PF can be roughly divided into two parts, conduits and information chunks. Conduits are channels for transferring and processing information chunks. The framework has four kinds of conduits: protocol, mux, factory and adapter. Each conduit, except adapter, has two sides (A and B), to which other conduits can be connected. Conduits are bi-directional, so that each side can send and receive information. [HJE95]

2.1 Conduits

- Protocol has one neighbour conduit on its both sides. It includes a finite state machine (FSM) and functions to produce and consume information chunks. Usually implementing a single layer in a protocol stack requires using several conduits that are specialized for specific functions.
- Mux is a conduit that has one conduit on its side A and any number of conduits on its side B. It is useful for separating different connections, for example. In that case there is an upper protocol instance for each connection above the mux, and they are created and deleted dynamically.
- Factory is used to install protocol instances on a mux. Factory has a prototype of the protocol, which it uses to create new protocol instances for new sessions on B-side of mux.
- Adapter is usually used to connect the framework to some interface, usually an application programming interface (API). Only its side A is connected to a conduit and side B implementation is for example an application or a hardware device.

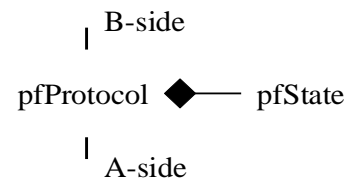


Figure 2. Protocol and state.

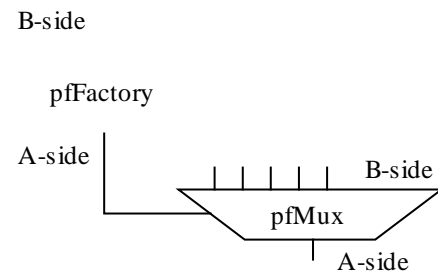


Figure 3. Mux and factory.

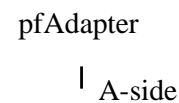


Figure 4. Adapter.

Note that `pfAdapter`, `pfFactory` and `pfMux` are all derived from `pfProtocol`, so they share the same basic behaviour. When `pfProtocol` is used as an example, the same example applies to other conduits as well. Usually `pfFactory` and `pfMux` are used without further inheritance, and it is not necessary to implement an FSM for them. The term “conduit implementation”, which is used in this document, refers to a class that is derived from `pfProtocol`, or an object instantiated from such class.

2.2 Using proxies

To connect conduit implementations to each other, `pfConduit` class is used to create proxies to them. This causes the usage of automatic memory management, which will count the references to the implementation. Creating `pfConduit` proxies increments `pfProtocol`'s reference counter. When a proxy is deleted, its destructor decrements the reference counter and checks it. If the counter has gone to zero, the proxy deletes the implementation. The idea of the proxies is to control access to the implementation, as the conduit implementations are used only through the proxies. Usually there should not be need for pointers straight to conduit implementations in the main program, only to proxies.

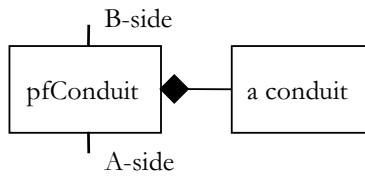


Figure 5. Every proxy has a conduit implementation.

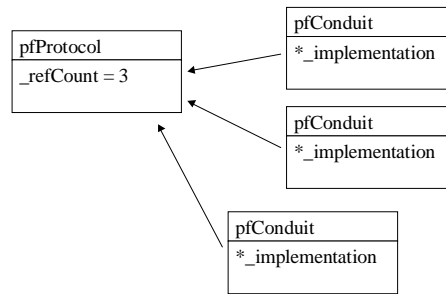


Figure 6. A conduit implementation can have many proxies.

```
// scoms/src/pf/conduit.h
// Basic interface for creating pfConduit proxies and connecting them.

class pfConduit
{
public:
    explicit pfConduit(pfProtocol *_implementation_ = 0);
    void connectToA(const pfConduit &conduit_);
    void connectToB(const pfConduit &conduit_);
    void disconnect(void);
}
```

Methods for sending information chunks to other conduits are available inside pfProtocol. Receiving information chunks is also implemented in pfProtocol, but it is invisible to designer (see chapter 4).

```
// scoms/src/pf/protocol.h
// The first two methods are most commonly used in protocol implementations.
// The last two are used by pf, and can be used for sending e.g.
// pfInstallTransporters in some rare cases.

class pfProtocol : public pfState, public sfTask, public pfTimers
{
public:
    virtual void toA(pfMessenger *messenger_);
    virtual void toB(pfMessenger *messenger_);
    virtual void toA(pfTransporter *transporter_);
    virtual void toB(pfTransporter *transporter_);
}
```

2.3 Information chunks

- Messenger is an entity containing information. It can be a protocol data unit (PDU), a service primitive or protocol's internal signal.
- Transporter is used to carry messengers or other information between conduits.
- Timeout is also a message that is received in protocol's FSM.

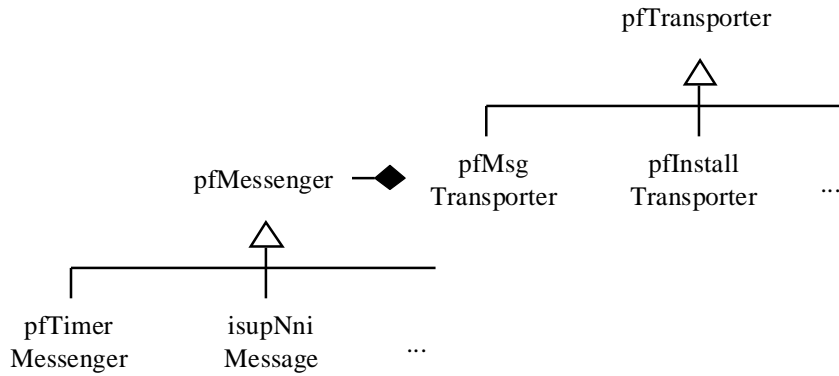


Figure 7. pfMsgTransporter carries messages.

2.4 Relations between classes

The following class diagrams show how the different classes of the framework interact. Use the graphs with the Reference guide together.

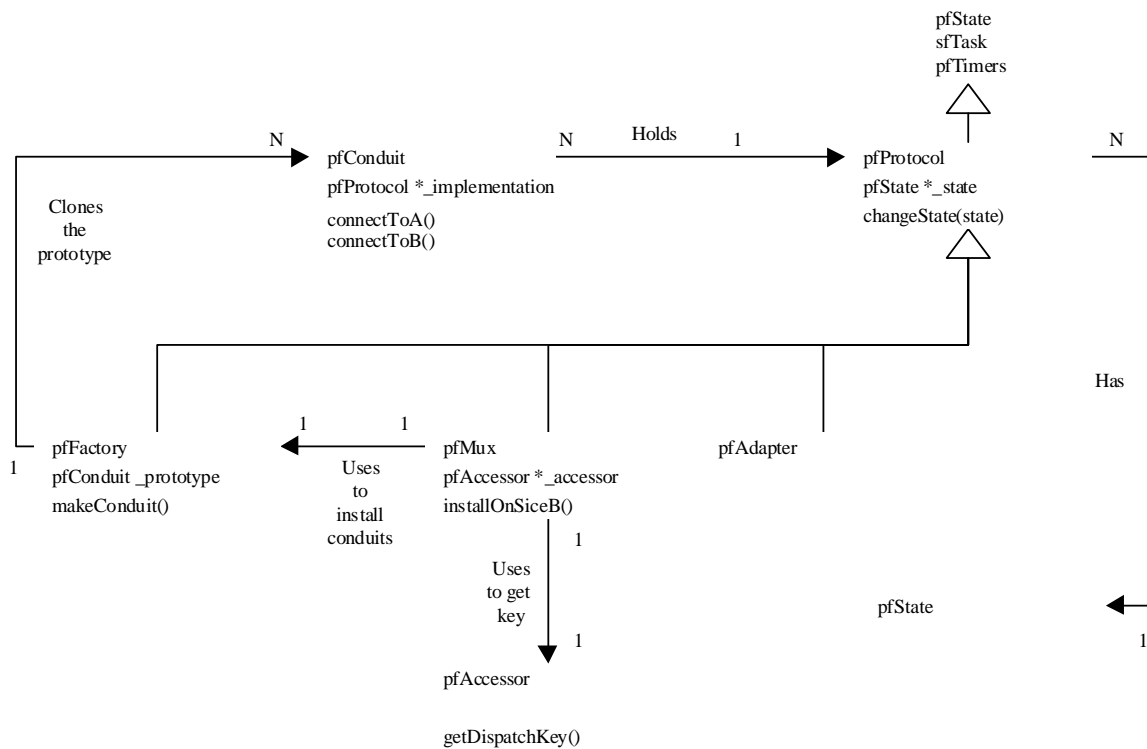


Figure 8. Relations between conduits (protocol, factory, mux and adapter).

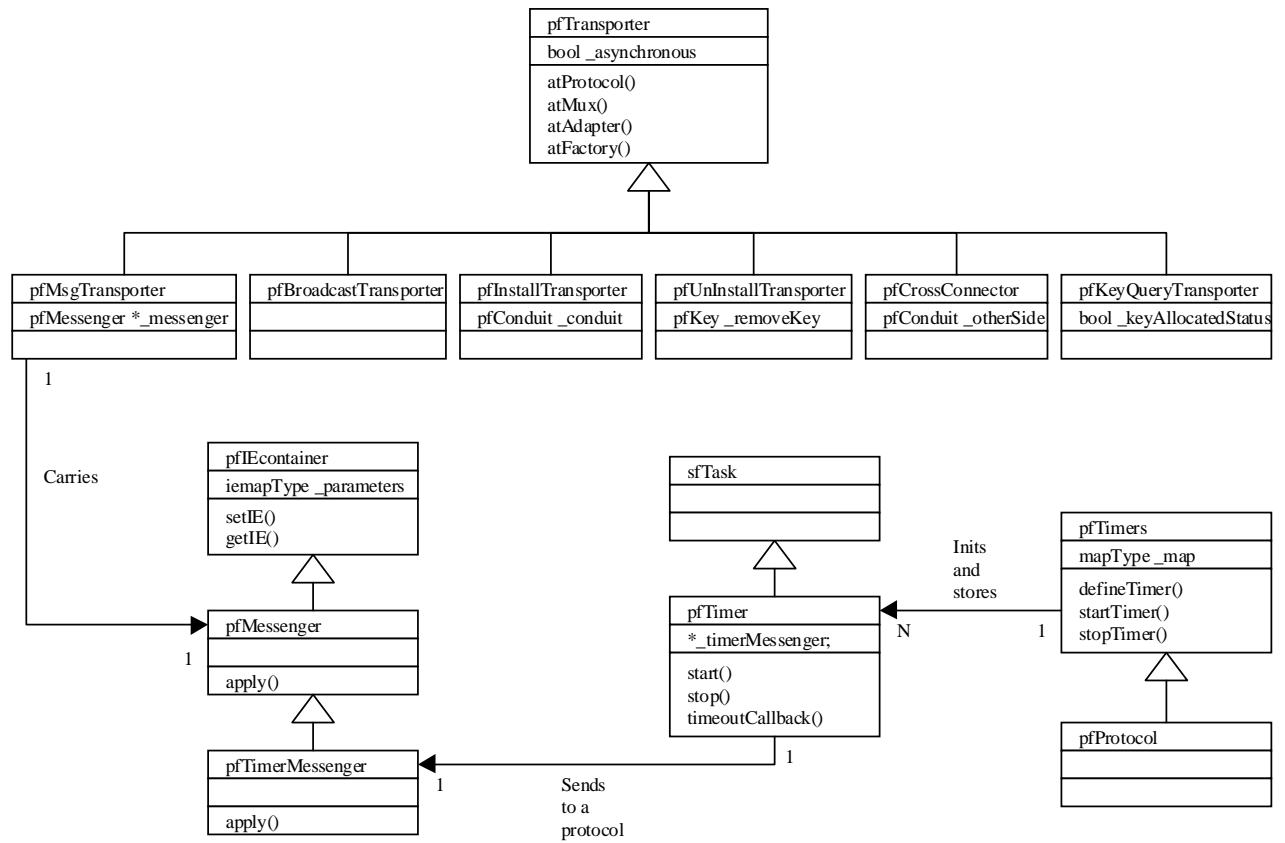


Figure 9. Relations between messages and timers.

3 REFERENCE GUIDE

This part of the document contains information about the most common classes in the PF module in alphabetical order.

3.1 pfAccessor

Every mux has an accessor to obtain a dispatch key (that specifies the connection) from an incoming message. The implemented accessor is derived from pfAccessor class, and it must contain at least getDispatchKey() method which is called by the mux (Figure 15, step 4). A maximum value for mux key must be provided when creating an accessor. The base class offers the method generateKey() for creating new, unallocated mux keys. PfAccessor implements the Strategy pattern [GHJV95, pages 315-323] (pfMux=Context, pfAccessor=Strategy). As a result the mux can be used anywhere in the conduit graph, because the accessor is the only protocol dependent part.

```
// scoms/src/protocol/isup/isupaccessor.cpp
// The getDispatchKey() method obtains the key that individuates the call.
// In ISUP it is Circuit Identification Code (CIC) which in practise is a
// timeslot, and here it is stored in the message as an Information Element
// (IE). See pfIEcontainer methods.

pfKey isupAccessor :: getDispatchKey(pfMessenger *messenger_)
{
    pfUlong timeSlot;
    if(messenger_>ieAvailable(ieCircuitIdentificationStr))
    {
        pfIE *ie = messenger_>getIE(ieCircuitIdentificationStr);
        ieCircuitIdentification *cic =
            dynamic_cast<ieCircuitIdentification*>(ie);
        THROW_IF_DYNAMIC_CAST_FAILED(cic);
        timeSlot = cic->getTimeSlot();
        debugPfUlong("Found timeslot ", timeSlot);
    }
    else
    {
        debugUser("Didn't find CIC in message!");
        THROW_INVALID_CALL_REFERENCE_VALUE;
    }

    return timeSlot;
}
```

3.2 pfAdapter

An adapter acts as an interface between the conduit world and some device or application. Some common rules about implementing an adapter for a device are sketched. The adapter is derived from the adapted device, for example an IP-socket, ATM-socket etc. pfDevice class has pure virtual methods readAction() and writeAction(), that are rewritten using the methods that the device offers. The readAction() method is called by pfDevice, when it has read a frame from the device, and it is given to the adapter for further processing. writeAction() is called when the device has processed a write request and the data has been sent. Information chunks arriving at adapter from A-side are received and handled in an FSM as usual.

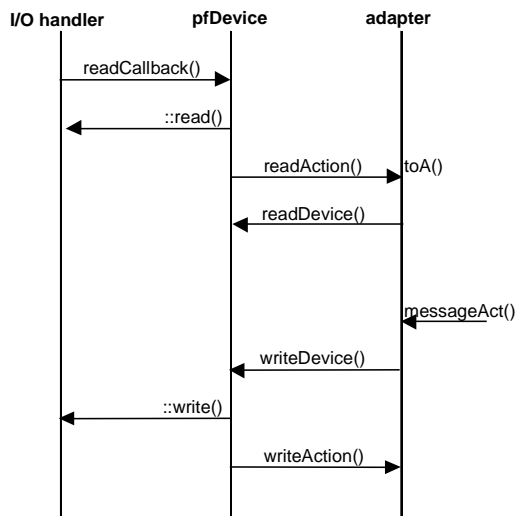


Figure 10. Reading and writing to socket.

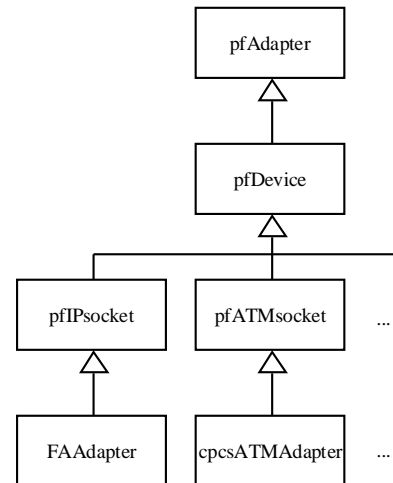


Figure 11. Adapter class hierarchy.

3.3 pfConduit

pfConduit is implemented using the Proxy pattern, which means that it provides a placeholder for another object, in order to control access to it. [GHJV95, pages 207-217] (pfConduit=Proxy, pfProtocol=RealSubject) When conduits are connected or disconnected using corresponding methods, the calls are actually redirected to the real conduit implementations. A common misconception is that pfProtocol is derived from pfConduit. This is not the case, the pfConduit class offers just a placeholder for a conduit implementation. The conduit implementations should be created using create() functions that return a pfConduit object. This causes the usage of the proxy memory management automatically, and it prevents outsiders from using the conduit's functions. This should be applicable at least in ordinary layer protocols, but some special cases may prevent it, for example if in main program the protocol's methods must be called. If that is the case, consider if it is possible to call these methods already in the create() function. If the implementation is Singleton, and the only instance is not supposed to be destroyed even if there are no proxies left, the pfProtocol's noReference() method must be overridden with one that never returns 0.

```

// Create a isup protocol and test adapter instances and connect them.

pfConduit *isupProxy = isupProtocol :: create();
pfConduit *adapterProxy = testAdapter :: create();
isupProxy->connectToB(adapterProxy);
adapterProxy->connectToA(isupProxy);
  
```

```

// A normal create function that returns a proxy.

pfConduit pfTestUpProtocol :: create(void)
{
    pfProtocol *protocol = new pfTestUpProtocol();
    pfConduit newConduit = pfConduit(protocol);

    return newConduit;
}
  
```

```
// A Singleton's create function could look like this:

pfConduit pfTestUpProtocol :: create(void)
{
    if(_only == 0)
    {
        _only = new pfTestUpProtocol();
    }
    pfConduit newConduit = pfConduit(_only);
    return newConduit;
}
```

3.4 pfDebug

Current events of the framework are automatically announced with the pfDebug trace. These events include changing state, receiving and sending primitives, as well as creating message and processing queued messages. pfDebug offers several public methods for different types of debug traces. The methods are used via macros, and output of them depends on the debug options that are set in the main program. Debugging must also be defined in makefile, and it can be set off for better performance. If the debugging is not defined, the macros are replaced with empty ones that do not cause function calls in vain. pfDebug prints the file and line number, from which the debug method is called. The trace must be enabled in the main program, using the definitions in debug.h. Usually debugOutputCout() is used, which prints the trace to screen.

```
debugUser("here we go");
debugPfUlong("number ", i);
debugString("string ", a);
```

In the next example a part of log is presented. Scheduled events start with timestamp and START_RUNNING line and end with END_RUNNING. In this case an ISUP protocol instance receives a setup request from upper layer, but fails to find a mandatory IE and throws an exception.

```
DEBUG: TRACE: TIMESTAMP: Thu Jul 22 12:38:13 1999 932636293 sec 464557 usec :
debug.cpp : 289
DEBUG: START_RUNNING: isupProtocol_258 runs sigSETUPreq_743 (isupIdle)
DEBUG: TRACE: CIC not present! : isupprotocol.cpp : 293
DEBUG:TRACE: isupProtocol::changeMuxReference found no CIC in message! :
isupprotocol.cpp : 219
DEBUG: TRACE: occured exception = string : sigException : exception.cpp : 114
DEBUG: TRACE: file = string : isupprotocol.cpp : exception.cpp : 115
DEBUG: TRACE: line = pfUlong : 220 : exception.cpp : 116
DEBUG: TRACE: cause = pfUlong : 41 : exception.cpp : 117
DEBUG: TRACE: Failed to set correct mux reference! : isupstate_idle.cpp : 71
DEBUG: END_RUNNING
```

3.5 pfException

Exceptions should be used for error handling. pfException class provides methods for recovering information about the class and the occurrence that causes throwing the exception. Using an exception requires using PF_EX_INFO macro that tells the exception's constructor the point where the exception was thrown. In pf/error.h there are various macros defined especially for using pfException methods. The exceptions should be caught when there is, for example, some protocols specific action that must be done in error situation. If they are not caught before, the pfProtocol's runCallback() finally catches them.

3.6 pfFactory

pfFactory implements the Prototype pattern which uses a prototypical instance to create new objects [GHJV95, pages 107-116] (pfConduit=Prototype, pfFactory=Client). When creating a factory, it must be given a protocol prototype as a parameter, after which it is connected to a mux. The factory is connected on B-side of mux. It is a default conduit for incoming messages, i.e. if there is no connection corresponding to the dispatch key, the message is sent to factory (Figure 15, step 6).

```
// scoms/src/sw/swpdhlink.cpp
// ISUP prototype and factory

pfConduit isupProxy = swSwitch::instance()->getPDHprototype();
isupProxy.setId(linkIdentifier_);
_factoryProxy = pfFactory::createFactory(isupProxy);
_factoryProxy.setId(linkIdentifier_);
```

```
// scoms/src/sw/swlink.cpp

_factoryProxy.connectToA(_muxProxy);
_muxProxy.connectToB(_factoryProxy);
```

3.7 pfFrame

pfFrame is used as a data storage. The frame data is handled octets at a time. If more elaborate methods are needed, use pfBitString class that handles data as bits.

```
// scoms/src/pf/frame.h
// Some important methods of the frame explained.

public:

// Returns the length of the data in octets.
pfUlong length(void) const;

// Methods for reading/writing data to frame. Read data is removed from the
// frame.
void putFirst(pfByte byte_);
void putFirst16bit(pfUlong value_);
void putFirst24bit(pfUlong value_);
void putFirst32bit(pfUlong value_);
void putFirst(const pfByte *byte_, pfUlong length_);
pfByte getFirst(void);
pfUlong getFirst16bit(void);
pfUlong getFirst24bit(void);
pfUlong getFirst32bit(void);

// The data can be also written to the end of frame.
void putLast(pfByte byte_);
void putLast16bit(pfUlong value_);
void putLast24bit(pfUlong value_);
void putLast32bit(pfUlong value_);
void putLast(const pfByte *byte_, pfUlong length_);
pfByte getLast(void);
pfUlong getLast16bit(void);
pfUlong getLast24bit(void);
pfUlong getLast32bit(void);

// These methods read the data at a given offset, but do not remove it.
pfByte read(pfUlong offset_) const;
pfUlong read16bit(pfUlong offset_) const;
pfUlong read24bit(pfUlong offset_) const;
pfUlong read32bit(pfUlong offset_) const;
```

```
// The data can also be written to specific offset. The original data will
// be replaced. getSubFrame() creates a new frame, but does not alter the
// original data.
void write(pfByte value_, pfUlong offset_);
pfFrame getSubFrame(pfUlong start_, pfUlong length_) const;
```

3.8 pfIE

pfIE is base class for information elements. IE is usually a parameter in a PDU, and it can contain any information needed. As the pfMessenger is derived from pfIEcontainer, it can be used for storing information elements. The information elements must contain clone() methods, that are used by pfIEcontainer.

```
// scoms/src/protocol/isup/isupprotocol.cpp
// pfMessenger is used to carry information elements. In this example the
// same circuit identification code IE is set to every message.

void isupProtocol :: setCallReference(pfMessenger *message_)
{
    if (_CICset == true)
    {
        pfIE *iePtr = _CIC->clone();
        message_->setIE(ieCircuitIdentificationStr, iePtr);
    }
    else
    {
        debugUser("CIC not known and not set to message");
    }
    return;
}
```

3.9 pfIEcontainer

pfIEcontainer is used to store information elements. It uses a Standard Template Library (STL) map to store the IE instances, that are identified with strings. pfIEcontainer provides methods for storing and deleting information elements that are derived from pfIE class.

3.10 pfMessenger

pfMessenger represents a message or an event. The apply() method is used to invoke correct action in protocol's state machine, so it must be rewritten in each message. The apply() method is called from pfTransporter as a consequence of scheduler's action (for example in Figure 15, step 12). As the current state of protocol is given as parameter to apply() method, the correct Act() method can be called in the FSM. pfMessenger implements Command pattern, which means that when someone makes a request to an object, the request is passed to another object by handing it the command object [GHJV95, pages 233-242] (pfMessenger=Command, pfState=Receiver). This way any conduit can handle the messenger by passing it along to a neighbour conduit or another object. Here the messenger is passed to a state object.

```
// scoms/src/protocol/isup/isupnnimessages.cpp
// Current state of ISUP is given as parameter to apply().

void isupNniIAMpdu :: apply(pfState *state_, pfProtocol *protocol_)
{
    isupNniMessageInputs *input =
        dynamic_cast<isupNniMessageInputs*>(state_);
```

```

        THROW_IF_DYNAMIC_CAST_FAILED(input);
        input->isupNniIAMpduAct(this, protocol_);
        return;
    }

```

3.11 pfMux

The multiplexer works together with a factory and an accessor. The accessor is used to obtain a dispatch key from a message. When the key is available, the mux checks from its STL map structure if there is a connection matching the key or not. If there is, the message is sent to correct protocol, and if not, a new protocol instance is created using the factory (Figure 15, steps 1-6). When creating a mux, it is given the accessor as parameter. Important: every conduit implementation that is installed on mux, must contain a `cloneImplementation()` method which is called by the factory.

```

// scoms/src/sw/swpdhlink.cpp
// An accessor is given to mux's constructor.

pfAccessor *accessor = new isupAccessor(_maxMuxKeyValue);
_muxProxy = pfMux::createMux(accessor);
_muxProxy.setId(linkIdentifier_);

```

```

// scoms/src/protocol/isup/isupprotocol.cpp

pfProtocol *isupProtocol :: cloneImplementation(void) const
{
    pfProtocol *protocol = new isupProtocol(*this);
    return protocol;
}

```

3.12 pfProtocol

pfProtocol is used to implement the behavior of a protocol using an FSM. The state machine is implemented using the State pattern, which means that each state is represented by a separate object. [GHJV95, pages 305-313] (pfProtocol=Context, pfState=State) Every pfProtocol instance has a pointer to its current state object, which is changed when a state change occurs. A protocol can change its state when it receives a message or when a timer expires (a timeout message is received). pfProtocol contains common functions for sending and receiving information chunks. A derived protocol has protocol specific data and methods, but it also delegates its behavior to state objects. The information chunks interact mainly with the state objects.

3.13 pfState

The FSM implements the Singleton pattern, meaning that instead of every protocol instance having its own state objects, there is only one instance of each state class in a binary [GHJV95, pages 127-134] (pfState=Singleton).

```

// scoms/src/protocol/isupstate_idle.cpp
// Here the usage of Singleton pattern is seen. The constructor is
// protected, and the static instance() method is used to get a pointer to
// the protocol instance. *_only is a private static pointer to the state,
// and if it is empty, a new (and only) state instance is created.

isupIdle *isupIdle :: _only = 0;

isupIdle *isupIdle :: instance(void)
{
    if (_only == 0)

```

```

    {
        _only = new isupIdle;
    }
    return _only;
}

isupIdle :: isupIdle(void)
    : isupState()
{
    return;
}
// ...

```

The state classes are not supposed to contain any protocol specific data. A state class must contain all the operations that the messengers invoke in it. The actions are implemented in *Act() methods, that are called by corresponding messages. Different states are supposed to receive different messages, so it is useful to implement default *Act() methods in a base state class, and only necessary methods in the specific state classes. The *Act() methods are usually represented in an abstract input class. A protocol that interfaces with another conduit using an abstract input class, must derive its own base state class from that input class.

```

// scoms/src/protocol/isup/isupstate.h
// The inputs are divided in 3 groups: primitives on up side, PDUs and
// timeouts. isupState is a base class, that contains default functions for
// inputs.

class isupState : public pfState,
                  public sigDownInputs,
                  public isupNniMessageInputs,
                  public isupTimeoutInputs
{
public:

    // sigDownInputs
    virtual void sigSETUPreqAct(
        sigSETUPreq *message_,
        pfProtocol *protocol_);
    //...

    // isupNniMessageInputs
    virtual void isupNniACMpduAct(
        isupNniACMpdu *message_,
        pfProtocol *protocol_);
    //...

    // Timeouts
    virtual void isupT1timeoutAct(pfProtocol *protocol_);
    virtual void isupT5timeoutAct(pfProtocol *protocol_);
    virtual void isupT7timeoutAct(pfProtocol *protocol_);
    virtual void isupT9timeoutAct(pfProtocol *protocol_);

protected:
    isupState(void);
    virtual ~isupState(void);

    isupProtocol *protocolCast(pfProtocol *protocol_) const;
};

```

Changing state is implemented in pfProtocol class. The changeState() method is then called from the FSM when needed. The changeState() method is given a pointer to a state instance as parameter. It is also important to change protocol's state to suitable one in its constructor.

```
// scoms/src/protocol/isupniprotocol.cpp
// ISUP's NI protocol constructor.

isupNiProtocol :: isupNiProtocol()
{
    changeState(isupNiState::instance());
    return;
}
```

3.14 pfSystem

pfSystem class encapsulates the scheduler framework from designer. When starting the system, the init() function must be called, with possible (command line) parameters. pfSystem also implements the Singleton pattern, and provides a global point of access to the only instance [GHJV95, pages 127-134] (pfSystem=Singleton).

```
// Init is called in the beginning of main program.
pfSystem::init(argc, argv);

// Run is called in the end of main program, after necessary conduits are
// created and connected. It causes an endless loop to begin.
pfSystem::instance()->run();

// Next runs only one step of sf scheduler. It can be used for debugging
// purposes.
pfSystem::instance()->next();
```

3.15 pfTimer

pfTimer is usually invisible to designer, and is used by the pfTimers class which is recommended for using timers. When a timeout occurs, the scheduler calls timeoutCallback() method in pfTimer. This causes sending a corresponding pfTimerMessenger to the protocol.

3.16 pfTimerMessenger

When a protocol receives a timeout message, it is handled just like other messages in the FSM, so the message's apply() method must be written to call the right *Act() method in the current state. The message must also contain a clone() method, that is used by pfTimer class.

```
// scoms/src/protocol/isuptimeouts.cpp

pfTimerMessenger *isupTltimeout :: clone(void) const
{
    isupTltimeout *messenger = new isupTltimeout(*this);
    return messenger;
}

void isupTltimeout :: apply(pfState *state_, pfProtocol *protocol_)
{
    isupTimeoutInputs *input = dynamic_cast<isupTimeoutInputs*>(state_);
    THROW_IF_DYNAMIC_CAST_FAILED(input);
    input->isupTltimeoutAct(protocol_);
    return;
}
```


3.17 pfTimers

pfTimers class, which is inherited to pfProtocol, is the interface for defining and using timers. When calling defineTimer() method, it is provided with an unique string, a timer message instance, and a timeout value. The class offers also methods for setting new timeout values, and checking if the timer is active or not.

```
// scoms/src/protocol/isup/isupprotocol.cpp
// isupTlStr is a string that specifies the timer.

defineTimer(isupTlStr, isupTltimeout::create(), _Tlvalue);
startTimer(isupTlStr);
stopTimer(isupTlStr);

// New timeout value can be set as milliseconds. A timer goes inactive when
// it is stopped or when it expires.

setTimeout(isupTlStr, 3000);
bool result = isTimerActive(isupTlStr);
```

3.18 pfTransporter

pfProtocol uses this class, when sending information chunks to other conduits, and the derived classes can be used in various way. The idea is to be able to send information between conduits without having to care what kind of conduit receives it or what the data is. A transporter is an implementation of the Visitor Pattern, which allows defining an operation in a conduit without changing the classes of the conduit [GHJV95, pages 331-344] (pfTransporter=Visitor, pfProtocol=Element). When a visitor (e.g. pfMsgTransporter) arrives at a conduit, the conduit performs an operation on the visitor that indicates the class of the conduit. If it is a mux, atMux() method is called, if it is protocol, atProtocol() is called, and so on.

- A message transporter (pfMsgTransporter) is used to carry a messenger (pfMessenger) around a conduit graph. Usually it originates either from a adapter or from a protocol.
- A broadcast transporter (pfBroadcastTransporter) is used to carry a messenger (pfMessenger) around a conduit graph. Usually it originates either from a protocol through a mux. The messenger must have a clone() method that is called when sending the transporters to protocols over mux.
- An install transporter (pfInstallTransporter) is used to install a conduit object to the B side of the first mux it encounters. Usually it originates from a factory.
- An un-install Transporter (pfUnInstallTransporter) is used to remove a conduit object from B side of the first mux in encounters. pfUnInstallTransporter's constructor must be given the mux key as parameter, so that the mux knows which conduit it must erase.

```
// scoms/src/protocol/isup/isupprotocol.cpp
// This is an example of changing the mux key by first uninstalling the
// connection, and then installing another one. Usually this kind of tricks
// are not necessary.

if(_key != timeSlot)
{
    pfUnInstallTransporter uninstaller =
        pfUnInstallTransporter::createUnInstallTransporter(_key);
    toA(&uninstaller);

    pfConduit conduit(this);
```

```

pfInstallTransporter installer =
    pfInstallTransporter::createInstallTransporter(conduit);
installer.setKey(timeSlot);
toA(&installer);
}

```

- A cross connector (pfCrossConnector) is used to establish an interconnection between two protocol conduits in different links. It has the other side conduit as a data member.
- A key query transporter (pfKeyQueryTransporter) is used to query a mux if a specified key is allocated in its map or not.

```

// scoms/src/protocol/isup/isupniprotocol.cpp

pfBoolean isupNiProtocol :: isMuxKeyAllocated(const pfKey key_)
{
    pfKeyQueryTransporter keyQuery =
        pfKeyQueryTransporter::createKeyQueryTransporter(key_);
    toB(&keyQuery);
    pfBoolean result = keyQuery.keyAllocatedStatus();
    return result;
}

```

4 HOW PF AND SF WORK TOGETHER

pfTimer and pfProtocol are derived from sfTask which is a base class for schedulable objects. sfTask contains callback functions that are overridden in the derived classes. The protocol asks for processing time, and the timer asks for a timeout. The scheduler responds with calling the corresponding callback method.

4.1 Synchronous/asynchronous messages

A transporter can be set asynchronous or synchronous. If it is synchronous, it blocks other tasks and is processed immediately. If it is asynchronous, the conduit schedules it by requesting processing time from the scheduler.

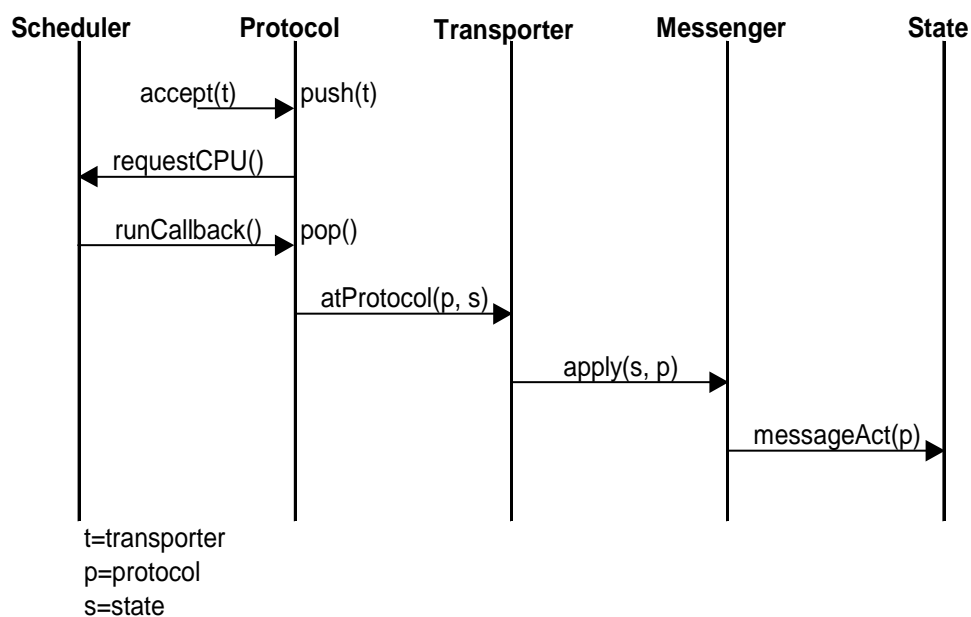


Figure 12. Scheduling an incoming message.

The `accept()` method is called by another conduit, through a proxy that holds the protocol. If the transporter was set synchronous, the `accept()` would cause `atProtocol()` call immediately without scheduling.

4.2 Synchronous/asynchronous conduits

Every conduit implementation has an `accept()` method that takes an incoming pfTransporter as parameter. That method then calls either `acceptSynchronous()` or `acceptAsynchronous()` methods. pfMux and pfFactory call `acceptSynchronous()` as default, while pfAdapter and pfProtocol check if the transporter is set to asynchronous or synchronous, and act on that basis. The asynchronous transporters are then pushed to scheduler's message queue. The transporter goes to the back of the queue as default, but if it is wanted to go first, the pfTransporter's `setSaveMethodHead()` method can be called. That could be usable if the protocol for example wanted to send a message to itself.

4.3 Timer function

Timer is invoked by a protocol, and the timeout action is much like with the messages. The timeout messages are set synchronous.

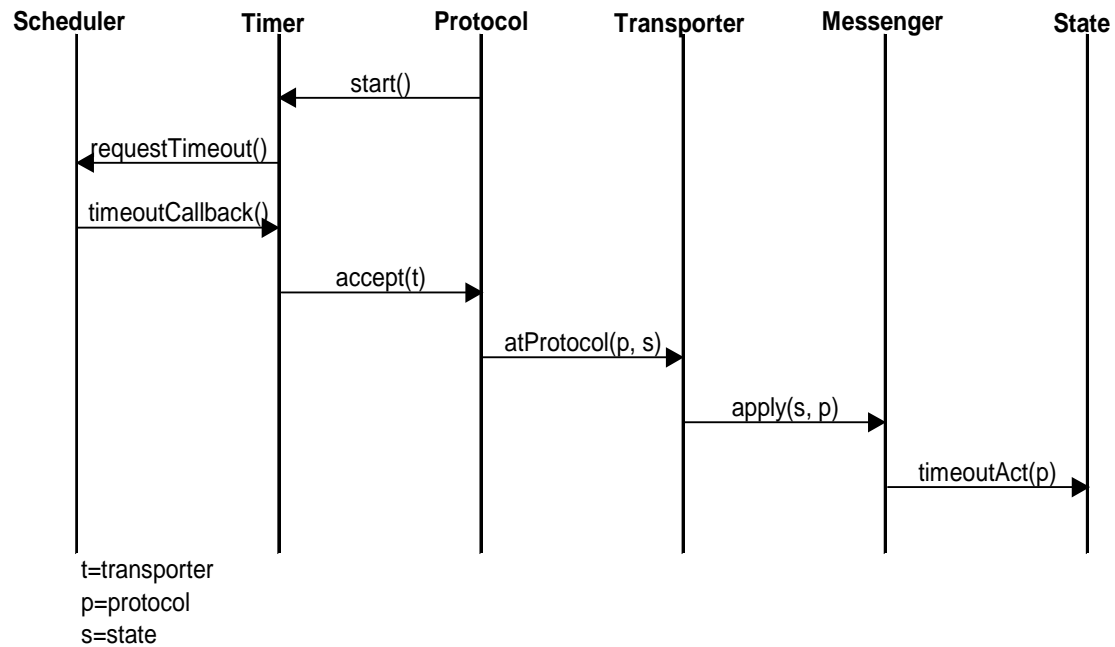


Figure 13. Scheduling a new timeout.

4.4 Function calls between conduits

Here is an example that should clarify the purpose and function of some important classes presented in the Reference guide. When a message arrives at a mux, an accessor obtains its call reference value. If the mux does not have such call yet, a new protocol instance is installed on it.

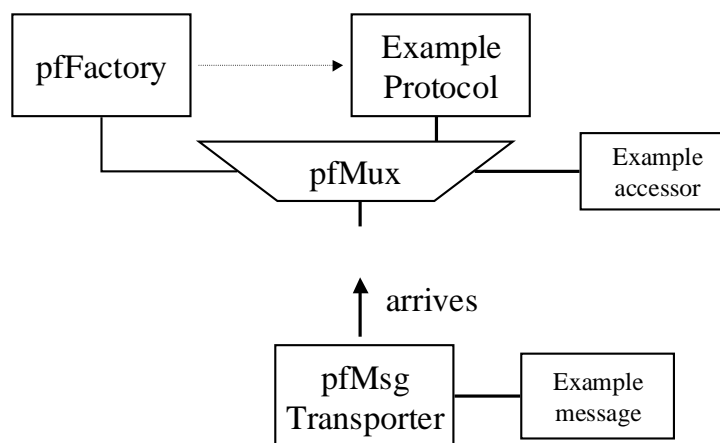


Figure 14. A message arrives at mux .

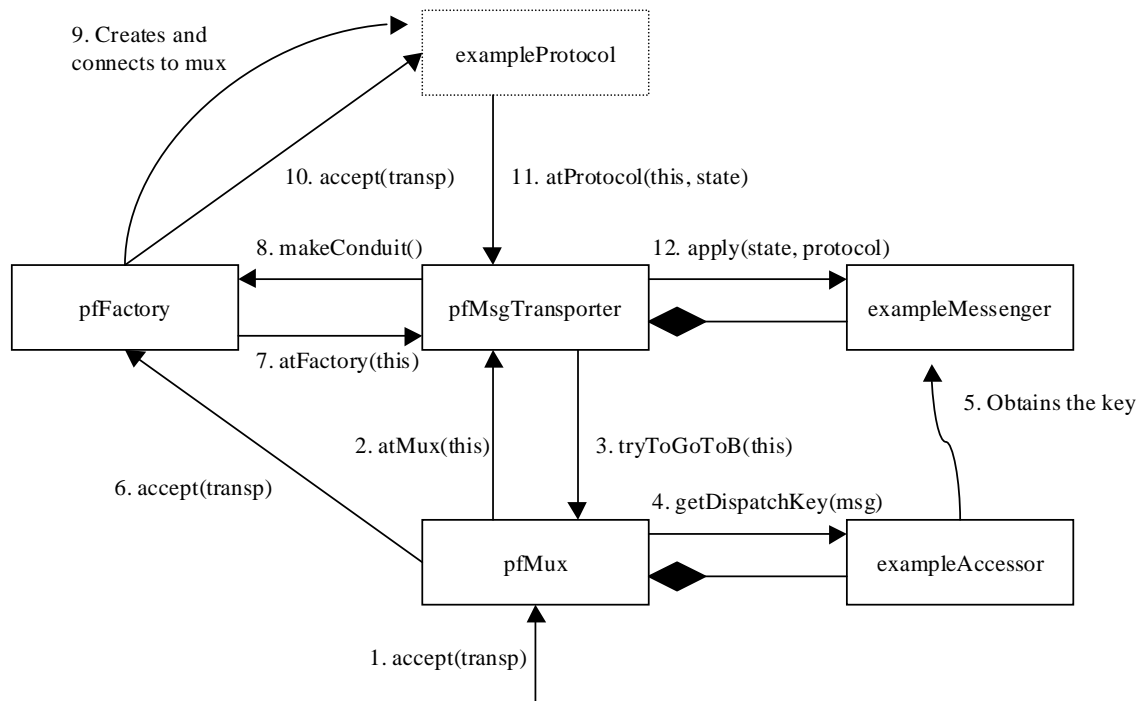


Figure 15. What happens when a new protocol object is installed on a mux. Steps 1-9 are handled synchronously.

5 USER'S GUIDE

This part of the document draws some common guidelines about using OVOPS++.

5.1 Some things to remember

- Create copy constructors to every class. This is especially important in classes like protocols that may be cloned, but it is easier to implement the copy constructors in every class.
- Conduits' `create()` methods should return a `pfProtocol` proxy. This is to enable the automatic memory management. Singleton classes must also have a `create()` method or similar.
- Use exceptions to handle errors, catch them when there is something you can do about it. The exceptions are caught eventually in `pfProtocol`'s `runCallback()` method, if not before.
- `clone()` method in following classes
 - Timeout messages (`pfTimerMessenger`). `pfTimer` class uses this when sending timeouts.
 - Information elements (`pfIE`). `pfIEcontainer` (all messages) uses this when copying information elements.
 - Messages carried with `pfBroadcastTransporters`. `pfBroadcastTransporter` clones the message at mux.
- `cloneImplementation()` methods in following classes
 - Every conduit that is installed on a mux. This is called when a factory installs new instances on the mux.
- Use `auto_ptr` only when it is possible that an exception is thrown, and a dynamically created object cannot thus be deleted. Remember to release the pointer if you want to save the object.

5.2 Main program

See an example of a main program in appendix 1. In the main program necessary conduits are created and connected, and the scheduler is started. Note `create()` functions, by which the conduit implementations are created. They return `pfConduit` objects, which prevents calling the conduit implementation's functions. In some rare cases it is necessary to call them, which means that the `create()` has to return a pointer to the conduit implementation. This can possibly be avoided by calling the functions in the `create()` function where the pointer is available. In such case a `pfConduit` proxy should be created right after creating the conduit implementation, to prevent a situation in which an exception is thrown and the pointer to the object is lost. Link id is not mandatory, it may be used to distinguish e.g. different ports.

5.3 Interface classes and FSM

States are derived from necessary interface classes that contain all the primitives, and pure virtual `*Act()` function declarations. These functions are defined in the base state class as default methods for receiving different kinds of messages. For example a protocol can inherit its up and down interfaces, as well as its timer inputs. In the derived state classes only the functions necessary for the specific state are overridden.

```

// scoms/src/protocol/isup/isupnnimessages.h
//
// This class contains pure virtual Act() methods for each of ISUP's
// messages. The class can be inherited at state classes, which get
// represented primitives as input messages. isupState class is derived from
// this.
//
class isupNniMessageInputs
{
public:
    virtual void isupNniACmpduAct(
        isupNniACmpdu *message_,
        pfProtocol *protocol_) = 0;
    virtual void isupNniANMpduAct(
        isupNniANMpdu *message_,
        pfProtocol *protocol_) = 0;
    virtual void isupNniCPGpduAct(
        isupNniCPGpdu *message_,
        pfProtocol *protocol_) = 0;

    // ...
};

```

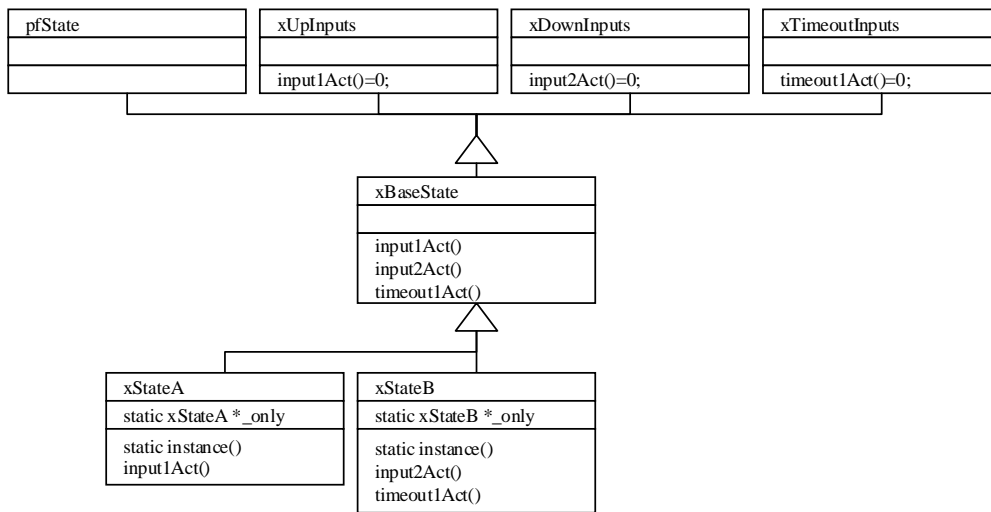


Figure 16. States are derived from input classes. Specific states redefine only the necessary functions.

6 EXAMPLE PROTOCOL: ISUP

This example presents ISUP protocol in Signalling System #7 (SS7) protocol stack. Lower layer is Message Transfer Part 3 (MTP-3) that offers transport service, and upper application that uses ISUP is Call Control (CC). The ISUP standard does not give much advice about implementation details, so this could be done in more than one ways. SCOMS implementation has two different protocol conduits, ISUP and network interface (NI), which handle different tasks. The NI protocol does encoding and decoding, i.e. transforms PDUs into messages and the opposite. Its FSM has only one state. The ISUP protocol has a complete FSM that describes the ISUP functionality.

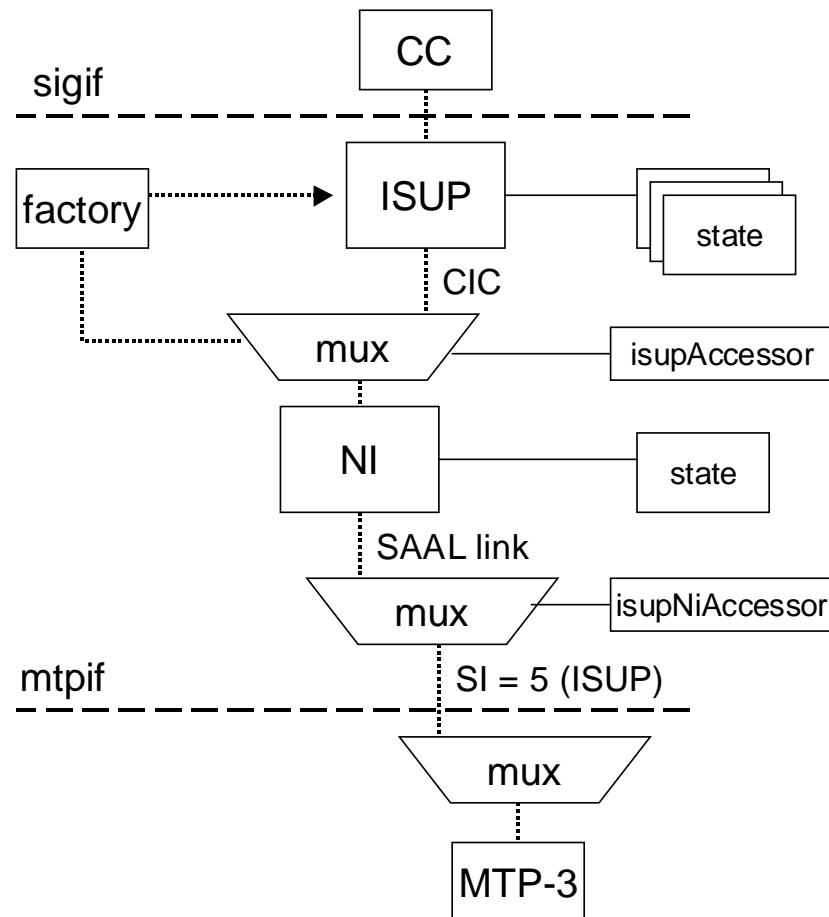


Figure 17. ISUP conduit diagram.

MTP-3 distinguishes its users by their Service Indicator (SI) value, that is 5 for ISUP. The next mux is for separating different ISUP links from each other, and the NI protocol is Singleton for each ISUP link. The different ISUP calls are specified by their CIC values, so there is one ISUP protocol instance (and CC instance) per each connection.

7 OVOPS++ EXERCISE

The purpose of this exercise is to familiarize the user with the common components and functions of OVOPS++. As the scheduling is not very visible to the user, the exercise concentrates on the PF module. The program can be used for testing PF and SF, for example searching memory leaks, simulating different situations and testing new features or changes in PF. You may find it useful in testing the OVOPS++ environment.

7.1 General description

pfTestAdapter simulates a stack on link level. The amount of loops to run is given to its constructor. Each loop creates a new connection by sending a pfTestSETUPind to the mux. The factory creates both protocols over the mux. When pfTestUpProtocol gets the pfTestSETUPind, it replies with a pfTestRELEASEreq that causes releasing the connection and deleting the conduits. See appendix 1 for an example implementation.

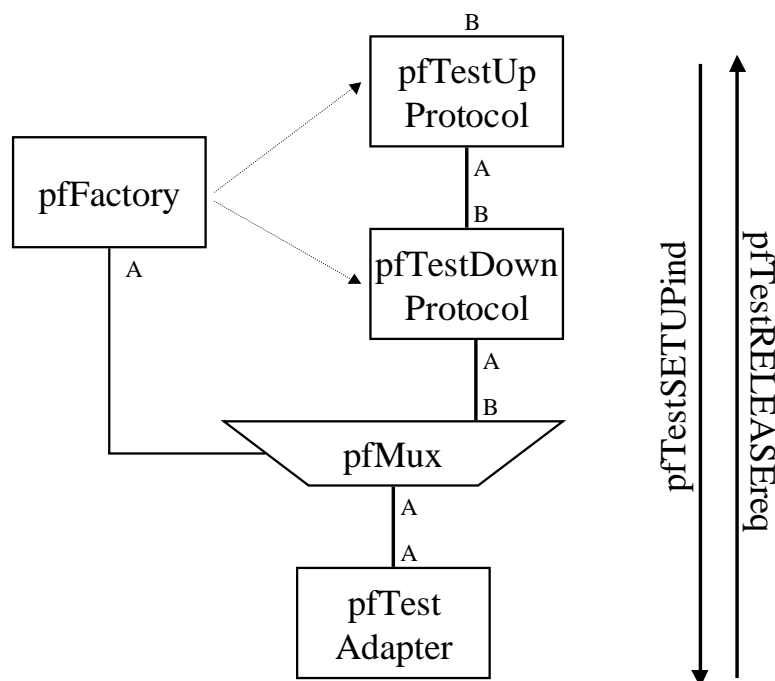


Figure 18. New connections are created in a loop over the mux.

7.2 Used PF classes

- pfConduit
- pfProtocol
- pfFactory
- pfMux
- pfAccessor
- pfAdapter
- pfMessenger
- pfUnInstallTransporter
- pfTimerMessenger

7.3 Implemented classes

- pfTestUpProtocol : public pfProtocol
- pfTestUpProtocolState : public pfState, public pfTestUpInputs
- pfTestDownProtocol : public pfProtocol
- pfTestDownProtocolState : public pfState, public pfTestUpInputs, public pfTestDownInputs, public pfTimeoutInputs
- pfTestDownProtocolIdle : public pfTestDownProtocolState
- pfTestDownProtocolActive : public pfTestDownProtocolState
- pfTestAccessor : public pfAccessor
- pfTestPrimitive : public pfMessenger
- pfTestSETUPind : public pfTestPrimitive
- pfTestRELEASEreq : public pfTestPrimitive
- pfTestUpInputs
- pfTestDownInputs
- pfTestTimeoutInputs
- pfTestTimeout : public pfTimerMessenger
- pfTestAdapter : public pfAdapter
- pfTestAdapterState : public pfState, public pfTestDownInputs, public pfTestTimeoutInputs

7.4 Finite state machines

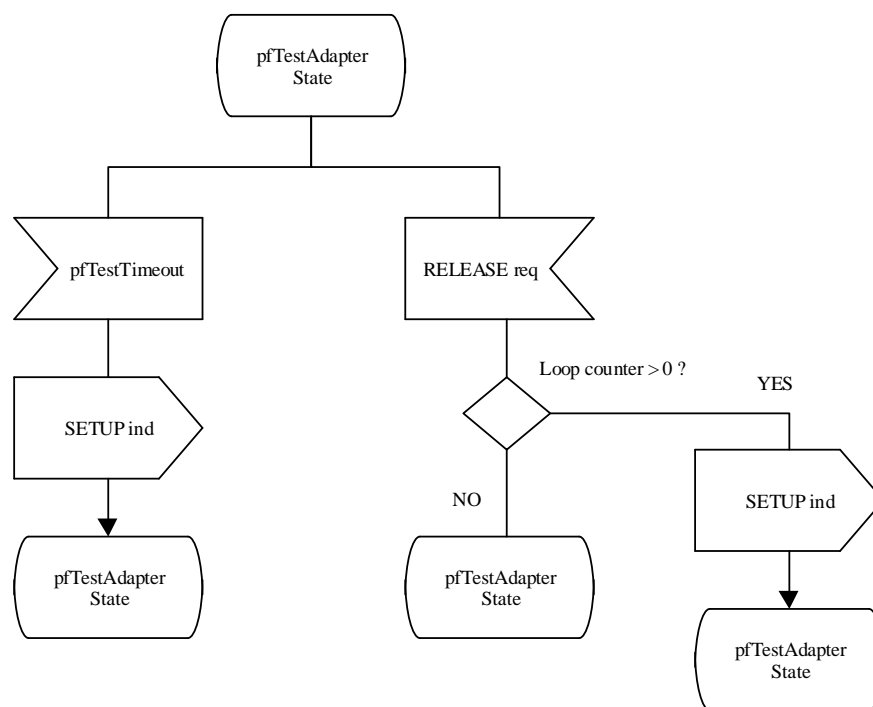


Figure 19. Adapter's FSM. The timer is started in the adapter's constructor, and its expiration causes sending the first message.

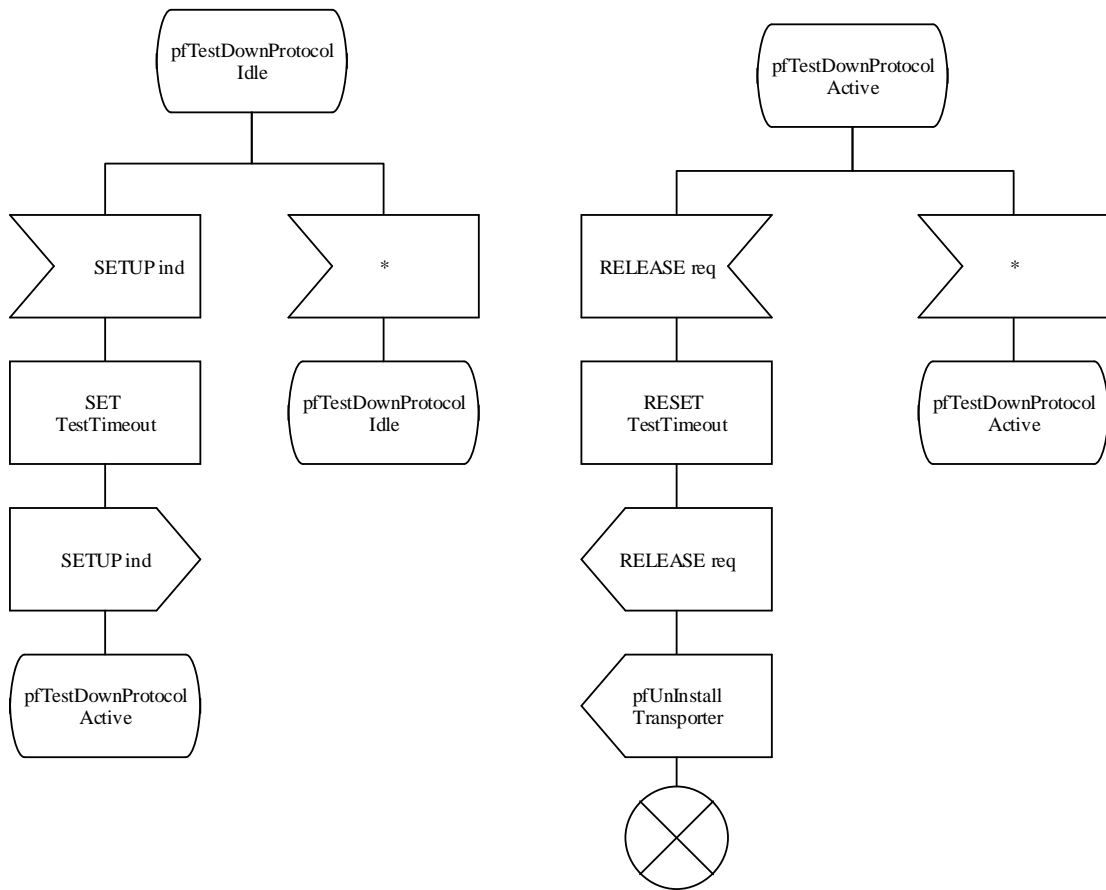


Figure 20. DownProtocol's FSM

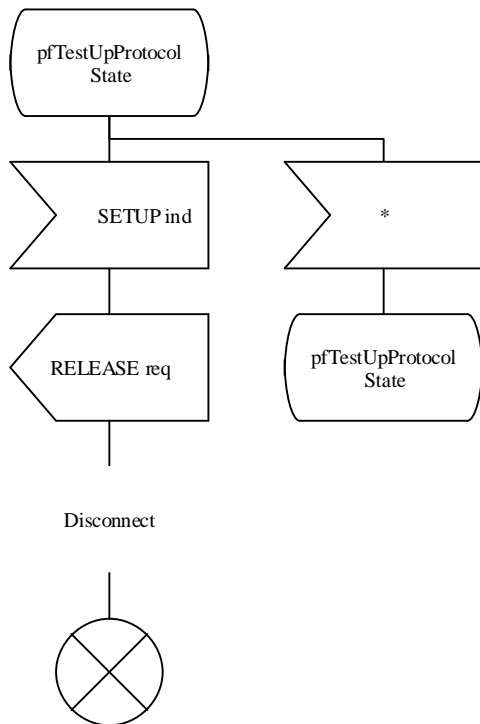


Figure 21. UpProtocol's FSM

7.5 Class diagrams

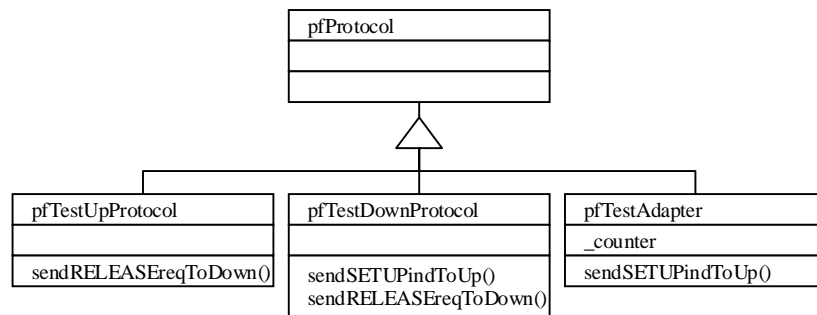


Figure 22. The protocols and the adapter.

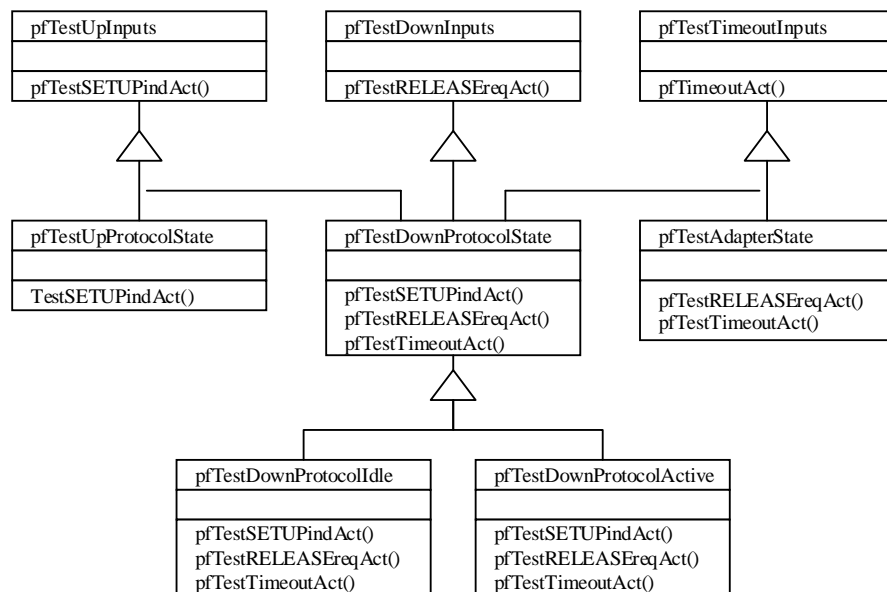


Figure 23. The states are derived from input classes and pfState.

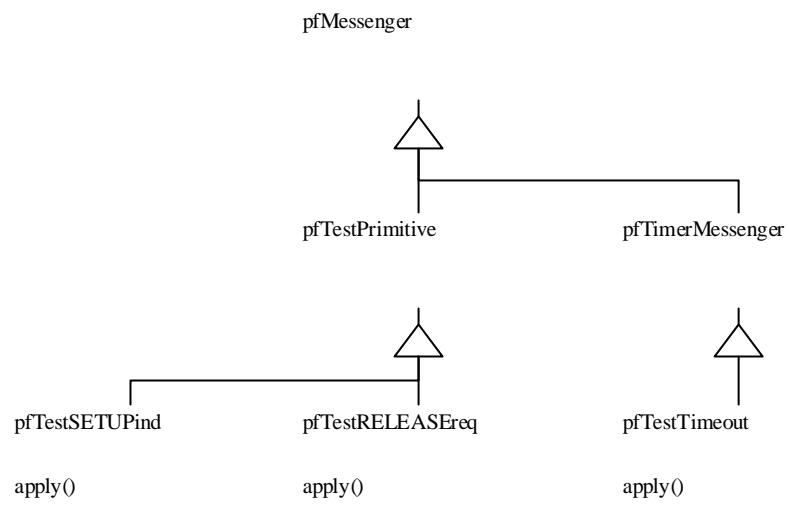


Figure24. The messages in this excercise.

REFERENCES

- [HJE95] Hermann Hüni, Ralph Johnson, Robert Engel. A Framework for Network Protocol Software. ACM Inc., 1995.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
- [MPR+96] Olli Martikainen, Vesa-Matti Puro, Juhana Räsänen, Timo Pärnänen, Pasi Nummisalo, Petteri Koponen. TOVE Project 1996 Deliverables.

APPENDIX 1.

```
//Editor-Info: -*- C++ -*-
//
//Subject: SCOMS project / exercise
//
//File: pftest.cpp
//
//Version: $Revision: 1.7 $
//
//State: $State: Exp $
//
//Date: $Date: 1999/04/07 09:13:25 $
//
//Organisation:
//    Helsinki University of Technology
//    Laboratory of Telecommunications Software and Multimedia
//
//Author:
//    Olli Suihkonen
//
//Description:
//    Main program of the OVOPS++ exercise
//
//Copyright:
//    Copyright 1999 Helsinki University of Technology
//    ALL RIGHTS RESERVED BETWEEN JANUARY 1996 AND JUNE 1999.
//
//Licence:
//
//
//History:

#include <typeinfo>

#include "testconduits.h"
#include "pf/mux.h"
#include "pf/factory.h"
#include "pf/debug.h"
#include "pf/system.h"

int main(int argc, char *argv[])
{
    // All debug printing directed to cout
    debugOutputCout();

    // Initialize system (sf, possible ORB etc.)
    pfSystem::init(argc, argv);

    // Initialize variables
    pfId id = 1; // link id, not mandatory
    int counter = 5; // counter value for loops
    pfUlong maxValue = 10000; // max instance (connections) above mux
    pfUlong timerForAdapter = 3000; // 3 secs. to start procedures

    // Create implementations (proxies)
    pfConduit adapterProxy = pfTestAdapter :: create(counter,
        timerForAdapter);
    pfConduit downprotocolProxy = pfTestDownProtocol :: create();
    pfConduit upprotocolProxy = pfTestUpProtocol :: create();

    // Create Factory (proxy), which will get the proxy of clonable protocol
    // Both prototypes are given to the factory, it installs and connects
    // them.
    pfConduit factory = pfFactory :: createFactory(downprotocolProxy,
        upprotocolProxy);
```

```

// Set link ids for proxies, also works without ids.
factory.setId(id);
adapterProxy.setId(id);

// Create mux (proxy) and its accessor (testAccessor)
pfTestAccessor *accessor = new pfTestAccessor(maxValue);
pfConduit mux = pfMux :: createMux(accessor);

// Connect conduits (adapter, mux, factory)
adapterProxy.connectToA(mux);
mux.connectToA(adapterProxy);
mux.connectToB(factory);
factory.connectToA(mux);

// Start system (sf)
try
{
    pfSystem::instance()->run();
}

// When the scheduler gets empty, sf throws an exception (if there are
// no file devices attached). pfSystem catches it and throws another
// one, which is caught here. If ORBacus is used, this doesn't happen.
catch (...)
{
    debugUser("Exception caught in main, the end.");
}
return 0;
}

```



```

//Editor-Info: -*- C++ -*-
//
//Subject: SCOMS project / exercise
//
//File: testconduits.h
//
//Version: $Revision: 1.6 $
//
//State: $State: Exp $
//
//Date: $Date: 1999/03/31 12:17:16 $
//
//Organisation:
//    Helsinki University of Technology
//    Laboratory of Telecommunications Software and Multimedia
//
//Author:
//    Olli Suihkonen
//
//Description:
//    OVOPS++ -exercice protocols and states, adapter and accessor.
//
//Copyright:
//    Copyright 1999 Helsinki University of Technology
//    ALL RIGHTS RESERVED BETWEEN JANUARY 1998 AND JUNE 2001.
//
//Licence:
//
//
//History:

#ifndef __TESTCONDUITS_H__
#define __TESTCONDUITS_H__

#include "pf/protocol.h"
#include "pf/adapter.h"
#include "pf/accessor.h"
#include "pf/error.h"
#include "testprimitives.h"
#include "testtimeouts.h"

// pfTestUpProtocol sends only release requests down. It needs
// cloneImplementation() method because it is installed on mux.
// Factory, that installs the protocols, calls that function.

class pfTestUpProtocol : public pfProtocol
{
public:
    static pfConduit create(void);
    virtual ~pfTestUpProtocol(void);
    pfProtocol *cloneImplementation(void) const;
    void sendRELEASEreqToDown(void);
protected:
    pfTestUpProtocol(const pfTestUpProtocol &other_);
private:
    pfTestUpProtocol(void);
};

// UpProtocol's only state must be able to receive inputs, that are defined
// in abstract pfTestUpInput class.

class pfTestUpProtocolState : public pfState,
                             public pfTestUpInputs
{
public:
    static pfTestUpProtocolState *instance(void);
    virtual ~pfTestUpProtocolState(void);
};

```

```

        virtual void pfTestSETUPindAct(
            pfTestSETUPind *primitive_,
            pfProtocol *protocol_);
    private:
        pfTestUpProtocolState(void);
        static pfTestUpProtocolState *_only;
};

// DownProtocol sends messages up and down. It also has a timer that expires
// if an error occurs and a release request is not received in time.

class pfTestDownProtocol : public pfProtocol
{
    public:
        static pfConduit create(void);
        virtual ~pfTestDownProtocol(void);
        pfProtocol *cloneImplementation(void) const;

        void initTimers(void);
        void releaseConnection(void);
        void sendSETUPindToUp(void);
        void sendRELEASEreqToDown(void);
    protected:
        pfTestDownProtocol(const pfTestDownProtocol &other_);
    private:
        pfTestDownProtocol(void);
};

// DownProtocol's states are derived from Up and Down inputs, because it is
// in the middle. The base class contains default methods, that just throw
// exceptions, so they are overridden in subclasses.

class pfTestDownProtocolState : public pfState,
                                public pfTestUpInputs,
                                public pfTestDownInputs,
                                public pfTestTimeoutInputs
{
    public:
        pfTestDownProtocolState(void);
        virtual ~pfTestDownProtocolState(void);

        virtual void pfTestSETUPindAct(
            pfTestSETUPind *primitive_,
            pfProtocol *protocol_);

        virtual void pfTestRELEASEreqAct(
            pfTestRELEASEreq *primitive_,
            pfProtocol *protocol_);

        virtual void pfTestTimeoutAct(
            pfTestTimeout *timeout_,
            pfProtocol *protocol_);
};

class pfTestDownProtocolIdle : public pfTestDownProtocolState
{
    public:
        static pfTestDownProtocolIdle *instance(void);
        pfTestDownProtocolIdle(void);
        virtual ~pfTestDownProtocolIdle(void);

        virtual void pfTestSETUPindAct(
            pfTestSETUPind *primitive_,
            pfProtocol *protocol_);

        virtual void pfTestRELEASEreqAct(
            pfTestRELEASEreq *primitive_,

```

```

        pfProtocol *protocol_);

        virtual void pfTestTimeoutAct(
            pfTestTimeout *timeout_,
            pfProtocol *protocol_);
    private:
        static pfTestDownProtocolIdle *_only;
};

class pfTestDownProtocolActive : public pfTestDownProtocolState
{
    public:
        static pfTestDownProtocolActive *instance(void);
        pfTestDownProtocolActive(void);
        virtual ~pfTestDownProtocolActive(void);

        virtual void pfTestSETUPindAct(
            pfTestSETUPind *primitive_,
            pfProtocol *protocol_);

        virtual void pfTestRELEASEreqAct(
            pfTestRELEASEreq *primitive_,
            pfProtocol *protocol_);

        virtual void pfTestTimeoutAct(
            pfTestTimeout *timeout_,
            pfProtocol *protocol_);
    private:
        static pfTestDownProtocolActive *_only;
};

class pfTestAccessor : public pfAccessor
{
    public:
        pfTestAccessor(pfUlong maxKeyValue_);
        virtual ~pfTestAccessor(void);

        // returns the key from message, or creates a new one.
        pfKey getDispatchKey(pfMessenger *messenger_);
};

// Adapter counts the connections that are created on the mux. A new
// connection is created when a setup indication is sent.

class pfTestAdapter : public pfAdapter
{
    public:
        static pfConduit create(int counter_, pfUlong startTime_);
        virtual ~pfTestAdapter(void);
        void decrementCounter(void);
        int getCounter(void) const;
        void sendSETUPindToUp(void);
    private:
        pfTestAdapter(int counter_, pfUlong startTime_);
        int _counter;
};

// Adapter has just one state.

class pfTestAdapterState : public pfState,
                            public pfTestDownInputs,
                            public pfTestTimeoutInputs
{
    public:
        static pfTestAdapterState *instance(void);
        pfTestAdapterState(void);

```

```
virtual ~pfTestAdapterState(void);

virtual void pfTestRELEASEreqAct(
    pfTestRELEASEreq *primitive_,
    pfProtocol *protocol_);

virtual void pfTestTimeoutAct(
    pfTestTimeout *timeout_,
    pfProtocol *protocol_);
private:
    static pfTestAdapterState *_only;
};

#endif // __TESTCONDUITS_H__
```

```

//Editor-Info: -*- C++ -*-
//
//Subject: SCOMS project / exercise
//
//File: testconduits.cpp
//
//Version: $Revision: 1.6 $
//
//State: $State: Exp $
//
//Date: $Date: 1999/03/31 12:17:16 $
//
//Organisation:
//    Helsinki University of Technology
//    Laboratory of Telecommunications Software and Multimedia
//
//Author:
//    Olli Suihkonen
//
//Description:
//    See the header file.
//
//Copyright:
//    Copyright 1999 Helsinki University of Technology
//    ALL RIGHTS RESERVED BETWEEN JANUARY 1998 AND JUNE 2001.
//
//Licence:
//
//
//History:

#include "testconduits.h"
#include "testprimitives.h"
#include "pf/debug.h"
#include "pf/error.h"
#include "pf/exception.h"
#include "pf/conduit.h"
#include <typeinfo>

// Singleton pattern implementation: pointers to only instances are set 0.
// Now the instance() methods can check if the object is created or not by
// checking the pointer.

pfTestUpProtocolState *pfTestUpProtocolState :: _only=0;
pfTestDownProtocolIdle *pfTestDownProtocolIdle :: _only=0;
pfTestDownProtocolActive *pfTestDownProtocolActive :: _only=0;
pfTestAdapterState *pfTestAdapterState :: _only=0;

//
// Class: pfTestUpProtocol
//

// Conduits should always be created with create() methods that return a
// pfConduit entity. If there is a need to call the conduit implementation's
// methods in main program, you should try to do it here instead.

pfConduit pfTestUpProtocol :: create(void)
{
    pfProtocol *protocol = new pfTestUpProtocol();
    pfConduit newConduit = pfConduit(protocol);
    return newConduit;
}

// A protocol's state must be set in its constructor.

pfTestUpProtocol :: pfTestUpProtocol(void)
    :pfProtocol()
{

```

```

        changeState(pfTestUpProtocolState::instance());
        return;
    }

    pfTestUpProtocol :: pfTestUpProtocol(const pfTestUpProtocol &other_)
        :pfProtocol(other_)
    {
        return;
    }

    pfTestUpProtocol :: ~pfTestUpProtocol(void)
    {
        return;
    }

    pfProtocol *pfTestUpProtocol :: cloneImplementation(void) const
    {
        pfTestUpProtocol *protocol = new pfTestUpProtocol(*this);
        return protocol;
    }

    void pfTestUpProtocol :: sendRELEASEreqToDown(void)
    {
        pfTestRELEASEreq *message = new pfTestRELEASEreq;
        toA(message);
        return;
    }

    //
    // Class: pfTestUpProtocolState
    //

    // Singleton pattern implementation: create the object only if there is not
    // one already.

    pfTestUpProtocolState *pfTestUpProtocolState :: instance(void)
    {
        if (_only == 0)
        {
            _only = new pfTestUpProtocolState;
        }
        return _only;
    }

    pfTestUpProtocolState :: pfTestUpProtocolState(void)
        :pfState(),
        pfTestUpInputs()
    {
        return;
    }

    pfTestUpProtocolState :: ~pfTestUpProtocolState(void)
    {
        return;
    }

    void pfTestUpProtocolState :: pfTestSETUPindAct(
        pfTestSETUPind *,
        pfProtocol *protocol_)
    {
        pfTestUpProtocol *protocol = dynamic_cast<pfTestUpProtocol*>(protocol_);
        THROW_IF_DYNAMIC_CAST_FAILED(protocol);
        protocol->sendRELEASEreqToDown();
        protocol->disconnect();
        return;
    }
}

```

```

//
// Class: pfTestDownProtocol
//

pfConduit pfTestDownProtocol :: create(void)
{
    pfProtocol *protocol = new pfTestDownProtocol();
    pfConduit newConduit = pfConduit(protocol);
    return newConduit;
}

pfTestDownProtocol :: pfTestDownProtocol(void)
    :pfProtocol()
{
    changeState(pfTestDownProtocolIdle::instance());
    initTimers();
    return;
}

pfTestDownProtocol :: ~pfTestDownProtocol(void)
{
    return;
}

pfTestDownProtocol :: pfTestDownProtocol(const pfTestDownProtocol &other_)
    :pfProtocol(other_)
{
    initTimers();
    return;
}

pfProtocol *pfTestDownProtocol :: cloneImplementation(void) const
{
    pfTestDownProtocol *protocol = new pfTestDownProtocol(*this);
    return protocol;
}

void pfTestDownProtocol :: initTimers(void)
{
    defineTimer(TestTimeoutstr, pfTestTimeout :: create(),
        TestTimeoutvalue);
    return;
}

void pfTestDownProtocol :: sendSETUPindToUp(void)
{
    pfTestSETUPind *message = new pfTestSETUPind;
    toB(message);
    return;
}

void pfTestDownProtocol :: sendRELEASEreqToDown(void)
{
    pfTestRELEASEreq *message = new pfTestRELEASEreq;
    toA(message);
    return;
}

//
// Class: pfTestDownProtocolState
//

pfTestDownProtocolState :: pfTestDownProtocolState(void)
    :pfState(),
    pfTestUpInputs(),
    pfTestDownInputs(),
    pfTestTimeoutInputs()
{

```

```

        return;
    }

pfTestDownProtocolState :: ~pfTestDownProtocolState()
{
    return;
}

// These are the default functions for Down protocol's FSM. Here they just
// throw exceptions, in the derived states their functionality is different.

void pfTestDownProtocolState :: pfTestSETUPindAct(
    pfTestSETUPind *,
    pfProtocol *)
{
    throw pfException(PF_EX_INFO);
    return;
}

void pfTestDownProtocolState :: pfTestRELEASEreqAct(
    pfTestRELEASEreq *,
    pfProtocol *)
{
    throw pfException(PF_EX_INFO);
    return;
}

void pfTestDownProtocolState :: pfTestTimeoutAct(
    pfTestTimeout *,
    pfProtocol *)
{
    throw pfException(PF_EX_INFO);
    return;
}

//
// Class: pfTestDownProtocolIdle
//

pfTestDownProtocolIdle *pfTestDownProtocolIdle :: instance(void)
{
    if (_only == 0)
    {
        _only = new pfTestDownProtocolIdle;
    }
    return _only;
}

pfTestDownProtocolIdle :: pfTestDownProtocolIdle(void)
    :pfTestDownProtocolState()
{
    return;
}

pfTestDownProtocolIdle :: ~pfTestDownProtocolIdle(void)
{
    return;
}

void pfTestDownProtocolIdle :: pfTestSETUPindAct(
    pfTestSETUPind *,
    pfProtocol *protocol_)
{
    pfTestDownProtocol *protocol =
        dynamic_cast<pfTestDownProtocol *>(protocol_);
    THROW_IF_DYNAMIC_CAST_FAILED(protocol);
    protocol->sendSETUPindToUp();
}

```



```

        protocol->startTimer(TestTimeoutstr);
        protocol->changeState(pfTestDownProtocolActive::instance());
        return;
    }

    void pfTestDownProtocolIdle :: pfTestRELEASEreqAct(
        pfTestRELEASEreq *primitive_,
        pfProtocol *protocol_)
    {
        return;
    }

    void pfTestDownProtocolIdle :: pfTestTimeoutAct(
        pfTestTimeout *,
        pfProtocol *)
    {
        debugUser("Timeout arrived at downProtocolIdle");
        return;
    }

    //
    // Class: pfTestDownProtocolActive
    //

    pfTestDownProtocolActive *pfTestDownProtocolActive :: instance(void)
    {
        if (_only == 0)
        {
            _only = new pfTestDownProtocolActive;
        }
        return _only;
    }

    pfTestDownProtocolActive :: pfTestDownProtocolActive(void)
        :pfTestDownProtocolState()
    {
        return;
    }

    pfTestDownProtocolActive :: ~pfTestDownProtocolActive(void)
    {
        return;
    }

    void pfTestDownProtocolActive :: pfTestSETUPindAct(
        pfTestSETUPind *primitive_,
        pfProtocol *protocol_)
    {
        return;
    }

    // When a release request arrives at DownProtocol's active state, it sends
    // a pfUnInstallTransporter and sends it to the mux to release the
    // connection. A conduit implementation has its mux key as a data member.

    void pfTestDownProtocolActive :: pfTestRELEASEreqAct(
        pfTestRELEASEreq *primitive_,
        pfProtocol *protocol_)
    {
        pfTestDownProtocol *protocol =
            dynamic_cast<pfTestDownProtocol *>(protocol_);
        THROW_IF_DYNAMIC_CAST_FAILED(protocol);
        protocol->stopTimer(TestTimeoutstr);
        pfKey key = protocol->getKey();
        protocol->sendRELEASEreqToDown();
        pfUnInstallTransporter uninstaller =
            pfUnInstallTransporter::createUnInstallTransporter(key);
        protocol->toA(&uninstaller);
    }

```

```

    return;
}

void pfTestDownProtocolActive :: pfTestTimeoutAct(
    pfTestTimeout *,
    pfProtocol *)
{
    debugUser("Timeout arrived at pfTestDownProtocolActive");
    return;
}

//
// Class: pfTestAccessor
//

pfTestAccessor :: pfTestAccessor(pfUlong maxKeyValue_)
    : pfAccessor(maxKeyValue_)
{
    return;
}

pfTestAccessor :: ~pfTestAccessor(void)
{
    return;
}

pfKey pfTestAccessor :: getDispatchKey(pfMessenger *messenger_)
{
    pfKey muxRef = generateKey();
    return muxRef;
}

//
// Class: pfTestAdapter
//

pfConduit pfTestAdapter :: create(int counter_, pfUlong startTime_)
{
    pfProtocol *adapter = new pfTestAdapter(counter_, startTime_);
    pfConduit newConduit = pfConduit(adapter);
    return newConduit;
}

pfTestAdapter :: pfTestAdapter(int counter_, pfUlong startTime_)
    :pfAdapter()
{
    changeState(pfTestAdapterState::instance());
    _counter = counter_;
    pfTimerMessenger *timeout = pfTestTimeout :: create();
    defineTimer(TestTimeoutstr, timeout, startTime_);
    startTimer(TestTimeoutstr);
    return;
}

pfTestAdapter :: ~pfTestAdapter(void)
{
    return;
}

void pfTestAdapter :: decrementCounter(void)
{
    _counter--;
    return;
}

int pfTestAdapter :: getCounter(void) const
{
    return _counter;
}

```

```

}

void pfTestAdapter :: sendSETUPindToUp(void)
{
    pfTestSETUPind *message = new pfTestSETUPind;
    toA(message);
    return;
}

//
// Class: pfTestAdapterState
//

pfTestAdapterState *pfTestAdapterState :: instance(void)
{
    if (_only == 0)
    {
        _only = new pfTestAdapterState;
    }
    return _only;
}

pfTestAdapterState :: pfTestAdapterState(void)
:pfState(),
  pfTestDownInputs(),
  pfTestTimeoutInputs()
{
    return;
}

pfTestAdapterState :: ~pfTestAdapterState(void)
{
    return;
}

void pfTestAdapterState :: pfTestRELEASEreqAct(
    pfTestRELEASEreq *,
    pfProtocol *protocol_)
{
    pfUlong value = 0;
    pfTestAdapter *adapter = dynamic_cast<pfTestAdapter*>(protocol_);
    THROW_IF_DYNAMIC_CAST_FAILED(adapter);
    value = adapter->getCounter();
    if(value > 0)
    {
        adapter->sendSETUPindToUp();
        adapter->decrementCounter();
    }
    return;
}

void pfTestAdapterState :: pfTestTimeoutAct(
    pfTestTimeout *,
    pfProtocol *protocol_)
{
    pfTestAdapter *adapter = dynamic_cast<pfTestAdapter*>(protocol_);
    THROW_IF_DYNAMIC_CAST_FAILED(adapter);
    adapter->sendSETUPindToUp();
    return;
}

```

```

//Editor-Info: -*- C++ -*-
//
//Subject: SCOMS project / exercise
//
//File: testprimitives.h
//
//Version: $Revision: 1.6 $
//
//State: $State: Exp $
//
//Date: $Date: 1999/03/31 12:17:16 $
//
//Organisation:
//    Helsinki University of Technology
//    Laboratory of Telecommunications Software and Multimedia
//
//Author:
//    Olli Suihkonen
//
//Description:
//    Primitives used in OVOPS++ -exercise.
//
//Copyright:
//    Copyright 1999 Helsinki University of Technology
//    ALL RIGHTS RESERVED BETWEEN JANUARY 1998 AND JUNE 2001.
//
//Licence:
//
//
//History:

#ifndef __TESTPRIMITIVES_H__
#define __TESTPRIMITIVES_H__

#include "pf/messenge.h"

// TestPrimitive is a base class for messages.

class pfTestPrimitive : public pfMessenger
{
public:
    pfTestPrimitive(void);
    virtual ~pfTestPrimitive(void);
};

// Setup indication goes up, and causes installing a new connection. Apply
// is the method that calls the correct method in the state.

class pfTestSETUPind : public pfTestPrimitive
{
public:
    pfTestSETUPind(void);
    virtual ~pfTestSETUPind(void);
    virtual void apply(pfState *state_, pfProtocol *const protocol_);
};

// Release request goes down and causes destroying the connection.

class pfTestRELEASEreq : public pfTestPrimitive
{
public:
    pfTestRELEASEreq(void);
    virtual ~pfTestRELEASEreq(void);
    virtual void apply(pfState *state_, pfProtocol *const protocol_);
};

// pfTestUpInputs contains methods for all messages that go up.

```

```
class pfTestUpInputs
{
    public:
        virtual void pfTestSETUPindAct(
            pfTestSETUPind *message_,
            pfProtocol *protocol_)=0;
};

// pfTestDownInputs contains methods for all messages that go down.

class pfTestDownInputs
{
    public:
        virtual void pfTestRELEASEreqAct(
            pfTestRELEASEreq *message_,
            pfProtocol *protocol_)=0;
};

#endif // __TESTPRIMITIVES_H__
```

```

//Editor-Info: -*- C++ -*-
//
//Subject: SCOMS project /
//
//File: testprimitives.cpp
//
//Version: $Revision: 1.6 $
//
//State: $State: Exp $
//
//Date: $Date: 1999/03/31 12:17:16 $
//
//Organisation:
//    Helsinki University of Technology
//    Laboratory of Telecommunications Software and Multimedia
//
//Author:
//    Olli Suihkonen
//
//Description:
//    See the header file.
//
//Copyright:
//    Copyright 1999 Helsinki University of Technology
//    ALL RIGHTS RESERVED BETWEEN JANUARY 1998 AND JUNE 2001.
//
//Licence:
//
//
//History:

#include "testprimitives.h"
#include "pf/error.h"
#include "pf/state.h"
#include <typeinfo>

//
// Class: pfTestPrimitive
//

pfTestPrimitive :: pfTestPrimitive(void)
    : pfMessenger()
{
    return;
}

pfTestPrimitive :: ~pfTestPrimitive(void)
{
    return;
}

//
// Class: pfTestSETUPind
//

pfTestSETUPind :: pfTestSETUPind(void)
    : pfTestPrimitive()
{
    return;
}

pfTestSETUPind :: ~pfTestSETUPind(void)
{
    return;
}

void pfTestSETUPind :: apply(pfState *state_, pfProtocol *protocol_)

```

```

{
    pfTestUpInputs *input = dynamic_cast<pfTestUpInputs*>(state_);
    THROW_IF_DYNAMIC_CAST_FAILED(input);
    input->pfTestSETUPindAct(this, protocol_);
    return;
}

//
// Class: pfTestRELEASEreq
//

pfTestRELEASEreq :: pfTestRELEASEreq(void)
    : pfTestPrimitive()
{
    return;
}

pfTestRELEASEreq :: ~pfTestRELEASEreq(void)
{
    return;
}

void pfTestRELEASEreq :: apply(pfState *state_, pfProtocol *protocol_)
{
    pfTestDownInputs *input = dynamic_cast<pfTestDownInputs*>(state_);
    THROW_IF_DYNAMIC_CAST_FAILED(input);
    input->pfTestRELEASEreqAct(this, protocol_);
    return;
}

```

```

//Editor-Info: -*- C++ -*-
//
//Subject: SCOMS project / exercise
//
//File: testtimeouts.h
//
//Version: $Revision: 1.6 $
//
//State: $State: Exp $
//
//Date: $Date: 1999/03/31 12:17:16 $
//
//Organisation:
//    Helsinki University of Technology
//    Laboratory of Telecommunications Software and Multimedia
//
//Author:
//    Olli Suihkonen
//
//Description:
//    pfTestTimeout class and its input class, OVOPS++ -exercise.
//
//Copyright:
//    Copyright 1999 Helsinki University of Technology
//    ALL RIGHTS RESERVED BETWEEN JANUARY 1998 AND JUNE 2001.
//
//Licence:
//
//
//History:

#ifndef __TESTTIMEOUTS_H__
#define __TESTTIMEOUTS_H__

class pfState;
class pfProtocol;

#include "pf/timer.h"
const string TestTimeoutstr = "testtimeout";
const int TestTimeoutvalue = 3000;

// pfTestTimeout is a message that is sent, when a timer expires. It must
// have a clone() method that returns a pfTimerMessage type pointer.

class pfTestTimeout : public pfTimerMessenger
{
public:
    static pfTimerMessenger *create(void);
    virtual ~pfTestTimeout(void);
    virtual void apply(pfState *state_, pfProtocol *protocol_);
    virtual pfTimerMessenger *clone(void) const;
protected:
    pfTestTimeout(const pfTestTimeout &other_);
private:
    pfTestTimeout(void);
};

class pfTestTimeoutInputs
{
public:
    virtual void pfTestTimeoutAct(
        pfTestTimeout *timeout_,
        pfProtocol *protocol_)=0;
};

#endif // __TESTTIMEOUTS_H__

```



```

//Editor-Info: -*- C++ -*-
//
//Subject: SCOMS project / exercise
//
//File: testtimeouts.cpp
//
//Version: $Revision: 1.6 $
//
//State: $State: Exp $
//
//Date: $Date: 1999/03/31 12:17:16 $
//
//Organisation:
//    Helsinki University of Technology
//    Laboratory of Telecommunications Software and Multimedia
//
//Author:
//    Olli Suihkonen
//
//Description:
//    See the header file.
//
//Copyright:
//    Copyright 1999 Helsinki University of Technology
//    ALL RIGHTS RESERVED BETWEEN JANUARY 1998 AND JUNE 2001.
//
//Licence:
//
//
//History:

#include "testtimeouts.h"
#include "pf/state.h"
#include "pf/error.h"

//
// Class: pfTestTimeout
//

pfTimerMessenger *pfTestTimeout :: create(void)
{
    pfTestTimeout *messenger = new pfTestTimeout;
    return messenger;
}

pfTestTimeout :: pfTestTimeout(void)
    :pfTimerMessenger()
{
    return;
}

pfTestTimeout :: pfTestTimeout(const pfTestTimeout &other_)
    :pfTimerMessenger(other_)
{
    return;
}

pfTestTimeout :: ~pfTestTimeout()
{
    return;
}

pfTimerMessenger *pfTestTimeout :: clone(void) const
{
    pfTestTimeout *messenger = new pfTestTimeout(*this);
    return messenger;
}

```

```
void pfTestTimeout :: apply(pfState *state_, pfProtocol *protocol_)
{
    pfTestTimeoutInputs *input = dynamic_cast<pfTestTimeoutInputs*>(state_);
    THROW_IF_DYNAMIC_CAST_FAILED(input);
    input->pfTestTimeoutAct(this, protocol_);
    return;
}
```

```

# Makefile for OVOPS++ test module.

# STATICLIB is used to specify an archive target - that is, the module
# object files are combined into one file with unix 'ar' program. This
# file is then easy to link into the binary in the top level makefile.
# You must write here a name for the archive file, for example libbisup.a.
STATICLIB =

# STATICBIN is used to specify a static binary target.
STATICBIN = static_test

# DYNBIN is used to specify a dynamically linked binary target.
DYNBIN = dynamic_test

### NOTE: STATICBIN MUST NOT BE SAME AS DYNBIN ###

# INSTALL is used in 'make install'.
INSTALL = install
INSTALLOPTS = -c -m 644

# INSTALLTARGET is directory where binaries are installed in 'make install'.
INSTALLTARGET = /tmp

# SUBDIRS is used to specify all directories (modules) which need
# to be compiled successfully to get all OWNMODULEFILES needed for
# linking to a binary file. Also used in deepclean and deepdep.
SUBDIRS =

# OWNSTATICLIBS specifies all library-files (*.a) in subdirs.
OWNSTATICLIBS =

# OWNDYNOBJS is used to specify all (dynamic) object files (*.o) in subdirs.
OWNDYNOBJS =

# OBJS must have all object files (*.o) of your module. It is necessary to
# define this and previous variables _before_ the next step
# (include Rules.Make), otherwise the makefiles WON'T WORK.
OBJS = testprimitives.o \
      testconduits.o \
      testtimeouts.o \
      pftest.o

#
# Include a Rules file
#
include $(OPPSRC)/Rules.Make

install : static dynamic
          $(INSTALL) $(INSTALLOPTS) $(STATICBIN) $(DYNBIN) $(INSTALLTARGET)

###
### NOTES:
###
### Uncomment for-lines below if you have SUBDIRS defined.
###
### If you get error messages from make, change variables in rule name
### (four variables in far left) to name you defined above. Or if
### they are undefined, comment them.
###

# Default 'static' rule for modules. Both STATICBIN and STATICLIB are
# specified so that this same rule would work for both individual rules and
# top level executable. One or the other will be empty, but it doesn't
# matter.
static : $(STATICBIN) $(STATICLIB)

# Default rule to make an archive target.
$(STATICLIB) : $(OBJS)

```

```

$(AR) $(AROPTIONS) $(STATICLIB) $(OBJS)
$(RANLIB) $(STATICLIB)
#   @for i in $(SUBDIRS); do ( cd $$i && $(MAKE) static ); done

# Default rule to make own static binary.
$(STATICBIN) : $(OBJS) $(STATICLIB)
#   @for i in $(SUBDIRS); do ( cd $$i && $(MAKE) static ); done
$(CC) $(CCFLAGS) $(INC_DIRS) -o $(STATICBIN) $(OBJS) $(OPP_LIBS)
$(OWNSTATICLIBS) $(ORBLIB) $(NAMELIB) $(EVENTLIB) $(PROPERTYLIB)

# Default 'dynamic' rule for modules. Both DYNTARGET and DYNBIN are
# specified so that this same rule would work for both individual rules and
# top level executable. One or the other will be empty, but it doesn't
# matter. Also OBJS is specified to avoid problems with possible
# subdirectories.
dynamic : $(DYNBIN) $(DYNLIB) $(OBJS)

# Default rule to make a dynamic library.
$(DYNLIB) : $(OBJS)
$(CC) -shared $(CCFLAGS) -o $(DYNLIB) $(OBJS)
#   @for i in $(SUBDIRS); do ( cd $$i && $(MAKE) dynamic ); done

# Default rule to make own dynamically linked binaries.
$(DYNBIN) : $(OBJS)
#   @for i in $(SUBDIRS); do ( cd $$i && $(MAKE) dynamic ); done
$(CC) $(CCFLAGS) $(INC_DIRS) -o $(DYNBIN) $(DYN_LIBS) $(OWNDYNOBJS)
$(OBJS)

# deepdep makes dependencies in current directory and all SUBDIRS.
deepdep : dep
#   @for i in $(SUBDIRS); do ( cd $$i && $(MAKE) deepdep ); done

# deepclean cleans current directory and all SUBDIRS.
deepclean : clean
#   @for i in $(SUBDIRS); do ( cd $$i && $(MAKE) deepclean ); done

# Dependencies are below. The line next to this is used by makedepend, so
# DO NOT DELETE

```