T-110.300 Protocol Specification

- Protocol Specification Theory
- Message Sequence Charts (MSC)
- Specification and Description Language (SDL)
- ASN.1, TTCN
- Protocol Engineering Process (PEP)
Motivation

- Telecom systems engineering is still a huge industry
  - networks, terminals, services
- Datacom networks consist of protocols
  - IP technologies today and in future are based on protocols
  - protocols exist between services and applications
  - also security systems include protocols
- Basic ideas about protocol systems are portable to every systems that communicate
  - independent of methods and languages
- Research and industry have use for the know how
  - research centers, software and hardware manufacturers, telecom and datacom operators, ISPs, content providers, ...
- *Open vacancies at VTT Information Technology, Nokia Research Center, Helsinki University and Helsinki University of Technology!*
Protocol engineering covers the whole life-cycle of a protocol:

- requirements are set during overall system design
- protocol design produces complete specification
- specification needs to be verified
- protocol implementation is derived from specification
- implementation must be tested thoroughly
Protocol engineering

History of protocol engineering

- First communication protocols were designed in late 60s (alternating bit protocol by Bartlett et al in 1969)
- Theory of protocol verification dates back to late 70s (IBM Zurich 1978)
- Special-purpose protocol design languages were developed during 80s
- The increase of CPU power has made analysis of moderate sized real life protocols feasible during last 5 years
- Protocol design and implementation tools and development environment products since early 90s
Protocol engineering as a discipline of its own
Protocol Specification

Overview

Protocol Architectures
Object Modeling
Signaling Procedures
Protocol Data
Protocol Behavior
Protocol Specification

Protocol architectures: designing layers

- allocate a well-defined function for each layer
- keep the number of layers to the minimum
- create a layer boundary at a point where the number of interactions is minimized
- design interlayer interactions as service primitives (confirmed and unconfirmed services)
- use layers to allow different levels of abstraction
- allow changes to layer functions or even change of a complete layer protocol without affecting other layers

=> set of functional **protocol entities** composing a stack
Signaling procedures

Interactions between protocol entities can be described as signaling procedures.

Description method for signaling procedures is Message Sequence Chart (MSC) Notation, standardized by ITU-T in Rec Z.120

Similar notation in UML: Sequence Diagrams

Software tools for editing MSC diagrams, e.g. SDT MSC editor
Message Sequence Chart

Graphical MSC notation

- entities = vertical lines with names
- signaling messages = arrows
- messages have names
- order of messages

MSC-diagram is not a complete description of the behavior of entities, typically only basic (successful) cases are described as MSCs
MSC - Example: GSM handover

- Handover_command
- Channel_activate
- Channel_activate_ack
- Handover_command
- Handover_access
- Physical_information
- Move_to_new_channel
- Handover_complete
- Handover_complete
The MSC diagrams from the previous step describe information as messages with names, but no detailed contents.

PDUs are derived from MSC diagrams as
- messages, which are composed of
- information elements, which are composed of
- parameters with specified data types

This representation is called the abstract syntax of PDUs

For transmission the PDU encoding is defined as
- exact encoding of each parameter data type as bits and bytes
- (optional) identifiers for information elements and PDUs
- exact bit and byte order
Designing PDUs

Design methods:
- the most common informal method is to use bit/byte maps, which make no distinction between abstract and transfer syntaxes
- abstract syntax definition languages: ASN.1, XDR, CORBA IDL
- encoding rules are associated to abstract syntax definition languages: Basic Encoding Rules (BER), Packed ERs (PER)

Design tools
- graphical message editors
- abstract syntax language tools, e.g. ASN.1-to-C translators like the Nokia CASN Compiler for ASN.1
- data definition within integrated design environments, e.g. SDT
Protocol entity behavior

The MSCs resulting from the system design are used as starting point for this step

Not only the basic scenarios but also error situations must be taken into account

Correctly behaving protocol entity must
- accept all correct sequences of input messages
- detect any incorrect sequence of messages
- recover from such protocol errors

Protocol entities are usually modeled as state automata
Finite State Automaton

FSA model of a protocol

(i) set of input elements $I$ and set of output elements $O$, i.e. PDUs and Abstract Service Primitives (ASPs)

(ii) set of states $S$ and state transition function $succ$:

$$
\begin{align*}
S \times I & \rightarrow S \quad \text{(input transitions)} \\
S \times O & \rightarrow S \quad \text{(output transitions)}
\end{align*}
$$
Extended Finite State Automaton

Basic FSA notation is usually extended by
- typed variables within state automaton (context variables)
- typed parameters within input and output messages
- conditions (predicates) for state transitions may be given as Boolean expressions
- transitions may include actions given as block statements
- hierarchical states

The resulting representation of protocol entity is extended finite state automaton (EFSA)

Example of a graphical EFSA notation is UML State Chart

Software tools for editing State Charts, e.g. SDT SC editor
Design criteria EFSA models

Correctly designed automaton must fulfill following design criteria

- guaranteed provision of service
- absence of unwanted behavior with no progress

Well-defined engineering practice is needed to meet these criteria

- design languages: UML, MSC, SDL and ASN.1
- design tools: SDT editors, simulator and validator
Protocol Engineering Process

• Two related processes

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>- Requirements</td>
<td>- Requirements</td>
</tr>
<tr>
<td>- Analysis</td>
<td>- Analysis</td>
</tr>
<tr>
<td>- Design</td>
<td>- Design</td>
</tr>
<tr>
<td>- Specification</td>
<td>- Implementation</td>
</tr>
<tr>
<td>- Verification</td>
<td>- Testing</td>
</tr>
<tr>
<td></td>
<td>- Maintenance</td>
</tr>
</tbody>
</table>

• Iterative, parallel

Specification

Implementation
Protocol Engineering Process

**Phases inside processes are also iterative**
Protocol Engineering Process

Summary of protocol engineering

Protocol life-cycle

- problem - results into requirements
- analysis - results into tentative solution
- design - results into specification
- verification - results into proved specification

----------

- implementation - results into executable software
- testing - results into final software

Methods and languages for each phase

- UML for requirements capturing and analysis
- MSC diagrams for signaling procedures
- SDL with ASN.1 for protocol entity design
- Reachability analysis for verification
- C code generation for software implementation
- TTCN with ASN.1 for software testing
MSC notation

- descriptive specification of communication behaviour of system components and their environment by means of message interchange (design phase)
- expressive trace language for visualization of observed behaviour (simulation and testing phases)

Notation has been standardized by ITU in Draft Rec Z.120

MSC may be given in two forms
- visual syntax with graphical representation
- program-like textual syntax
Message Sequence Chart

Areas of application

- descriptive specification of real-time systems, especially telecom networks and protocols
- specification of partial behaviour only

More formal design uses MSC diagrams as input

MSCs may be used in

- requirements specification
- interface specification
- validation, simulation and animation (tracing)
- test-case specification
- documentation
MSC building blocks

Basic MSC is built of elements like

- instances
- events
- messages

Structuring mechanisms for building more complex MSCs

- Decomposition and refining of an instance
- Composition of MSCs

All basic elements and structuring mechanisms have graphical symbol of their own.

A collection of MSCs describing a system are collected into an MSC document.
Abstract Syntax Notation One

An introduction to the ASN.1 language
ASN.1 - Background

• How to specify messages that are to be exchanged between two communicating systems (e.g. protocols)?

Message information contents is the same, but representation may differ

Common transmission format

Transmission media

System A

Message

System B

Message

01101001 10110101 01010010
ASN.1 - Background

- A message can be viewed from two abstraction levels

  **Abstract level**
  Logical message information contents = Abstract syntax

  **Concrete level**
  Representation during transmission = Transfer syntax
  Encoding 01101001 10110101  Decoding
  Representation inside systems = Local syntax
Abstract syntaxes for PDUs

• Abstract syntax of a PDU deals with
  – composition of a PDU from information elements
  – data types of information elements

• No attention to bits or bytes of the PDU being constructed
  – Responsibility of abstract syntax: Contents of a message
  – Responsibility of transfer syntax: Message representation

• Message transfer syntax is derived from message abstract syntax
ASN.1 - Basic concepts

- ASN.1 is a data type definition language

- Basic data concepts
  - Record => SEQUENCE type
  - List/array => SEQUENCE OF type
  - Mutually exclusive alternatives => CHOICE type
  - Primitive data types => ASN.1 primitive types
  - Constants => values

- Structuring
  - Package/module => module
ASN.1 - Basic concepts, cntd.

- Example:

```
MyModule MODULE DEFINITIONS ::= 
BEGIN
  MyType ::= INTEGER
  myValue INTEGER ::= 100
END
```

A module definition

A type definition.

A value definition

- ASN.1 is similar to type specification parts of programming languages
Intro - Summary

- Transfer syntax is needed for network data
- Network data is designed at abstract syntax level
- ASN.1
- BER, CER, DER, PER
Protocol Architectures: SDL

• Contents
  – Basics of SDL
    • Structure and Behaviour of Systems
    • Data in SDL
  – Building a Functional Model with SDL
  – Protocol Engineering with SDL
    • combining ASN.1
    • building protocol stacks
  – More advanced features of SDL
• Telechess protocol as an example
SDL

- Specification and Description Language
- Standardised by ITU-T (former CCITT)
- Recommendation Z.100, CCITT Blue Book
- Designed for systems engineering
- First version 1988, new version 1992 (object concepts)
- Formal language
- Can also be used for implementation
What to Describe with SDL

• Reactive systems: input - output
  – Telecommunication systems and protocols
• Discrete systems
  – Finite State Machines (FSM)
• Not suitable for
  – Continuous systems
    • steering a car
  – Heavy calculation
    • weather models
SDL is a formal language

- Unambiguous specification of systems
- Can be applied in many phases
  - specification
  - design
  - implementation
  - documentation
- Supports also informal specifications
- Formality does not say how to specify an arbitrary system
Overview of SDL

• SDL system is a model of a real world system
  – definition of system components
  – hierarchical model
  – Does not restrict the model, no absolute patterns

• a System consists of
  – blocks, that are connected with channels

• a Block consists of
  – processes, that are connected with signal routes
  – sub-blocks, that contain sub-blocks or processes

• a Process has
  – attributes defined with variables and external procedures
  – behaviour defined with EFSM, with states and transitions

• Types can be defined for
  – systems, blocks and processes
Overview ...

- Types can be used to
  - define *instance sets* and *instances*
  - to define subtypes with *inheritance*

- Topics not covered
  - *services*
  - *refinement*
  - *specialisation*
A System Consists of Blocks

- Hierarchical structures of an SDL system
- Blocks are connected with channels
  - channels vs. signal routes
- No behaviour
  - abstract structures, black boxes
- Usually contains signal and data type definitions
  - header kind of definitions
  - local name space
The Structure of the SDL system
The Structure of the SDL system

Block LocalStation

SIGNAL
OpenDoor,
DoorOpened,
Allocate(DoorNoType),
Allocated(Flid),
NotAllocated;

LtotE

DtoE

(DoorToEnv)

DoorControl

[OpenDoor,
Allocate]

DttoC

DoorOpened,
Allocated,
NotAllocated

Control

CtotCU

LtotCU

[FromCentral]

[ToCentral]

LtotE

PtotE

Display

EnvToPanel

PanelControl

PtotC

(ControlToPanel)
A System and its Environment

• Channels usually are connected also to the environment
  – otherwise no external input/output to/from the system

• Environment has processes
  – other parts of the real world system that are connected to the SDL system
    • e.g. users, large telecom system components
  – they are not specified with SDL

• Environment observes and controls the system
  – user interfaces, key pads, etc.
Processes

- Basic components modelling the real world system
- Processes have
  - attributes (data)
  - behaviour (state machine)
- Processes communicate
  - with each other and with their environment
  - using signals (or other similar mechanisms)
- Autonomous particles
- Behaviour <-> Signalling
- Signalling is asynchronous
- Communication is defined by
  - signals
  - signal lists
  - signal routes
The Structure of the SDL system
SDL System Behavior

- Behavior is defined for processes
  - blocks cannot have any behavior, they are only hierarchical building blocks

- Behavior is defined by using Extended Finite State Machines
  - $Extended = variables$ are used
  - SDL has its own graphical language for state machines

- Input-Action-Transition defines the behavior
  - a signal is received
  - some tasks are carried out, signal(s) sent out
  - transition to another state

- Conventional programming is possible
  - procedures
  - control structures
  - etc.
SDL Data and its usage

• Two levels
  – built-in features
  – new operators
• General data concepts and terminology
  – Values, Variables, Expressions, Literals, Operators
• SDL data types are ‘usual data types’
• History:
  – Derived from LOTOS (ISO 8807)
  – ACT-ONE model (defining operators axiomatically)
    • specifies what properties are
    • hard to master
• **Built-in features are usually adequate for specifications and protocol engineering**
Defining Attributes and Data

- Variable definition notation
  - resembles Pascal notation

- Predefined types
  - Integer
  - Character
  - Boolean
  - Pid (==process identifier)
  - Time
  - Duration

- Predefined generators
  - string, struct, array, powerset
  - e.g. charstring is defined using generators
Structs

- Normal record type
  - much like a C struct
- Exclamation mark (!) denotes fields

```plaintext
newtype MyStruct struct
    name Charstring;
    age Integer;
endnewtype MyStruct;

dcl record MyStruct;

class record!name = 'John'
```
Arrays

- Array(type IndexType, type ElementType)
- ‘Normal’ Array types
- No range defined!

```plaintext
newtype MyArray
    Array(Integer, Character)
endnewtype MyArray;
dcl chars MyArray;
chars(5):=‘i’
```
Strings

- Defines a list of values (or a set)
- Has a formal parameter: Emptylist
  - creates an empty list

```plaintext
newtype MyString
  String(MyRecord, Emptystring)
endnewtype MyString;
dcl set MyString;

set(5)!age := 27
```
Syntypes and Synonyms

• Used for defining more meaningful type names
  – e.g. for signal parameters
• Used for enumerated (lueteltu) types
  – e.g. for limited values
• Synonym is more like renaming
• Example:

```sql
syntype AgeType = Integer constants 0:120 endsyntype;
dcl myAge AgeType;
synonym Address Charstring;
```
Defining Behaviour

• Behaviour of the system can be divided in two
  – internal => processes
    • states and transitions
  – external => interacting with the environment
    • inputs and outputs
• Behaviour is defined using state machine language
  – Extended Finite State Machine (EFSM)
    • uses variables as an extension to ‘state space’
  – Graphical notation
Defining Behaviour

Initiating a process in start up (example)

- start-up symbol
- task symbol
  - can be used e.g. for process initialisation
- output symbol
- timer start
- state symbol
  - first real state
Defining Behaviour

State transitions

- input symbol
  - a signal from another process
  - a remote procedure call

- actions
  - tasks, output-symbols, timers, procedure calls, etc.

- new state
  - actual transition or to remain in the same state
Defining Behaviour

Control structures
- decision symbol
  - evaluation => decision
- in- and out-connectors
- procedure calls and macro outlets
- enables ‘normal’ programming structures with

```plaintext
NextDoor:= NextDoor+1

Allocate(NextDoor)
SetAllocated
WaitAllocated

Decision
False
Allocate(NextDoor)
SetAllocated
WaitAllocated

True
All_Allocated
Errorhandling

Connector
```
Modelling Time

- SDL with real time systems
  - Telecommunication systems especially
- Implementation phase
  - not usually necessary in specification phase
- Used for
  - controlling the release of limited resources
  - controlling the answers from unreliable peers
  - issuing actions on regular basis
Timers

- A stopwatch with alarm
- A timer instance is like a process
  - it is created, it dies and it can send a signal
- A timer can be set, examined and resetted

- \texttt{timer} \ heratys\_aamuksi(kelloTyyppi):=8; /*h*/
- \texttt{set} (10 /*h*/, heratys\_aamuksi(oma\_kello))
- \texttt{reset}(heratys\_aamuksi(oma\_kello))