Network Protocol Design and Evaluation

04 - Protocol Specification, Part II

Stefan Rührup

University of Freiburg
Computer Networks and Telematics
Summer 2009
Overview

- In Part I of this chapter:
  - Modeling with state machines and UML

- Part II:
  - Formal state machine models revisited
  - SDL - The specification and description language
  - Describing scenarios with Message Sequence Charts
Formal State Machine Models

- Limited expressiveness of FSMs
- UML state charts more powerful, but semantic variation points
- Formal semantics needed (esp. for validation)
- Extended state machine models and formal languages:
  - Communicating FSMs (addition: message queues)
  - Extended FSMs (addition: variables)
  - SDL (based on ext. FSMs, adds structural concepts)
Communicating FSMs

- Automata connected by bounded FIFO message queues (asynchronous communication)
- Input and output = send and receive
Communicating FSMs

- Automata connected by bounded FIFO message queues (asynchronous communication)

- Changes to the Mealy finite state machine model:
  - Input and output queues (finite)
  - Simplification of the transition function:
    - state transitions are triggered by either input or output (here called action), but not by both
    - closer to reality: send and receive operations are usually not coupled
  - finiteness is still maintained

[Holzmann 1991]
Example for Communicating FSMs (1)

- A simplified variant of the alternating bit protocol [Holzmann 1991]

  - Sender sends messages with a control bit, the *alternating bit*, indicated by msg0 and msg1.
  - Messages are acknowledged by the receiver, also using the alternating bit, indicated by ack0 and ack1.
  - After sending msg0, the sender expects ack0.
  - If it receives ack1 instead, msg0 is re-transmitted.
  - Notation: ! = send, ? = receive
  - We implicitly assume a single channel
Example for Communicating FSMs (2)

- CFM specification of the protocol [Holzmann 1991]
Example for Communicating FSMs (3)

- State transition table of the receiver protocol [Holzmann 1991]
Definition of a CFSM

[Holzmann 1991]

- A message queue is a triple \((S, N, C)\), where
  - \(S\) is a finite set called the queue/message vocabulary
  - \(N\) defines the size of the queue, and \(C\) its contents

- A communicating finite state machine is a tuple \((Q, q_0, M, T)\), where
  - \(Q\) is a finite, non-empty set of states,
  - \(q_0 \in Q\) is the initial state,
  - \(M\) is a set of message queues, and
  - \(T\) is a state transition relation, \(T: Q \times A \rightarrow Q\), where \(A\) is the set of actions (input, output, or \(\epsilon\))
Definition of a CFSM

[Holzmann 1991]

- The state transition relation maps a state and an action to a successor state.
- An action can be input, output or null action.
- We denote input actions by ? and output actions by !
  example: \( T(q_0, !\text{msg}) = q_1 \)
- Input and output actions change exactly one message queue
- \( T(q, a) = \emptyset \) unless otherwise specified
Extended FSMs

- So far...
  - Message exchange between CFSMs

- Still missing in CFSMs:
  - Variables
  - the ability to exchange arbitrary values
Extended FSMs

- Extensions to the CFSM model:
  - Variables (integer, finite range)
  - Queues can transfer integer values
  - Set of arithmetic and logical operators
Definition of an Extended FSM

[Holzmann 1991]

- A *extended finite state machine* is a tuple \((Q,q_0,M,V,T)\), where
  - \(Q\) is a finite, non-empty set of states,
  - \(q_0 \in Q\) is the initial state,
  - \(M\) is a set of message queues,
  - \(V\) is a set of variables, and
  - \(T\) is a state transition relation, \(T: Q \times A \rightarrow Q\),
    where \(A\) is the set of actions
    (input, output, *boolean conditions, assignments*, or \(\varepsilon\))
Extended FSMs, Example

```
in?x,y
(assume x≠0)
```

Assignments and conditions

```
q0 → q1:
y=y-x
q1 → q2:
x<y
q1 → q3:
y==0
q2 → q5:
out!x
q3 → q1:
x>y
q3 → q4:
x=x-y
q4 → q1:
x=x-y
```

[cf. Holzmann 1991]
Extended FSMs

- Formal model for specification of concurrent processes
- FSM minimization and combination can be applied here
- FSM minimization: Find an equivalent state machine with the minimum number of states
FSM Minimization

[Holzmann 1991]

Define an boolean array $E$ of dimension $|Q| \times |Q|$, init = false

for all entries $E[i,j]$

if the states $i$ and $j$ are defined for the same actions, set $E[i,j] := true$ (regardless of the next state)

end for

repeat

for all true entries $E[i,j]$

Check, if their next states are equivalent for all actions,
otherwise set $E[i,j] := false$

end for

until the number of false entries is not increased
**Step 1:** for all entries $E[i,j]$

if the states $i$ and $j$ are defined for the same actions, set $E[i,j] := true$ (regardless of the next state)
Step 2: for all true entries $E[i,j]$  
Check, if their next states are equivalent for all actions,  
i.e. $\forall a \ E[T(i,a),T(j,a)]$, otherwise set $E[i,j] := \text{false}$

### State transition table $T$

<table>
<thead>
<tr>
<th>State</th>
<th>In</th>
<th>Out</th>
<th>Next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>msg1</td>
<td>-</td>
<td>$q_1$</td>
</tr>
<tr>
<td>$q_0$</td>
<td>msg0</td>
<td>-</td>
<td>$q_2$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>-</td>
<td>ack1</td>
<td>$q_3$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>-</td>
<td>ack0</td>
<td>$q_0$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>msg0</td>
<td>-</td>
<td>$q_4$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>msg1</td>
<td>-</td>
<td>$q_5$</td>
</tr>
<tr>
<td>$q_4$</td>
<td>-</td>
<td>ack0</td>
<td>$q_0$</td>
</tr>
<tr>
<td>$q_5$</td>
<td>-</td>
<td>ack1</td>
<td>$q_3$</td>
</tr>
</tbody>
</table>

### Equivalence table $E$

<table>
<thead>
<tr>
<th>Action $a$</th>
<th>$T(q_0,a)$</th>
<th>$T(q_3,a)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg1</td>
<td>$q_1$</td>
<td>$q_5$</td>
</tr>
<tr>
<td>msg0</td>
<td>$q_2$</td>
<td>$q_4$</td>
</tr>
</tbody>
</table>

- $E[q_1,q_5] = 1$
- $E[q_2,q_4] = 1$
**FSM Minimization**

**Step 2: for all** true entries $E[i,j]$  
Check, if their next states are equivalent for all actions,  
i.e. $\forall a \ E[T(i,a),T(j,a)]$, otherwise set $E[i,j] := false$

<table>
<thead>
<tr>
<th>State</th>
<th>In</th>
<th>Out</th>
<th>Next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>msg1</td>
<td>-</td>
<td>$q_1$</td>
</tr>
<tr>
<td>$q_0$</td>
<td>msg0</td>
<td>-</td>
<td>$q_2$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>-</td>
<td>ack1</td>
<td>$q_3$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>-</td>
<td>ack0</td>
<td>$q_0$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>msg0</td>
<td>-</td>
<td>$q_4$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>msg1</td>
<td>-</td>
<td>$q_4$</td>
</tr>
<tr>
<td>$q_4$</td>
<td>-</td>
<td>ack0</td>
<td>$q_0$</td>
</tr>
<tr>
<td>$q_5$</td>
<td>-</td>
<td>ack1</td>
<td>$q_3$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>In</th>
<th>Out</th>
<th>Next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_1$</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$q_2$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$q_3$</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$q_4$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$q_5$</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>In</th>
<th>Out</th>
<th>Next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>0</td>
<td>q1</td>
<td></td>
</tr>
<tr>
<td>$q_1$</td>
<td>0</td>
<td>q0</td>
<td></td>
</tr>
<tr>
<td>$q_2$</td>
<td>0</td>
<td>q0</td>
<td></td>
</tr>
<tr>
<td>$q_3$</td>
<td>1</td>
<td>q0</td>
<td></td>
</tr>
<tr>
<td>$q_4$</td>
<td>0</td>
<td>0</td>
<td>q1</td>
</tr>
<tr>
<td>$q_5$</td>
<td>0</td>
<td>1</td>
<td>q0</td>
</tr>
</tbody>
</table>

Equivalence table $E$

<table>
<thead>
<tr>
<th>State</th>
<th>In</th>
<th>Out</th>
<th>Next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$q_1$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$q_2$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$q_3$</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$q_4$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$q_5$</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Action $a$

$T(q_2,a)$

$T(q_4,a)$

$E[q_0,q_0] = 1$
FSM Minimization

**Step 2: for all** true entries $E[i,j]$

Check, if their next states are equivalent for all actions, i.e. $\forall a \ E[T(i,a),T(j,a)]$, otherwise set $E[i,j] := false$

State transition table $T$

<table>
<thead>
<tr>
<th>State</th>
<th>In</th>
<th>Out</th>
<th>Next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>msg1</td>
<td>-</td>
<td>$q_1$</td>
</tr>
<tr>
<td>$q_0$</td>
<td>msg0</td>
<td>-</td>
<td>$q_2$</td>
</tr>
<tr>
<td>$q_1$</td>
<td></td>
<td>ack1</td>
<td>$q_3$</td>
</tr>
<tr>
<td>$q_2$</td>
<td></td>
<td>ack0</td>
<td>$q_0$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>msg0</td>
<td>-</td>
<td>$q_4$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>msg1</td>
<td>-</td>
<td>$q_5$</td>
</tr>
<tr>
<td>$q_4$</td>
<td></td>
<td>ack0</td>
<td>$q_0$</td>
</tr>
<tr>
<td>$q_5$</td>
<td></td>
<td>ack1</td>
<td>$q_3$</td>
</tr>
</tbody>
</table>

Equivalence table $E$

<table>
<thead>
<tr>
<th>State Transition Table $T(q_i,a)$</th>
<th>$T(q_j,a)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$q_1$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$q_2$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$q_3$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$q_4$</td>
</tr>
<tr>
<td>$q_4$</td>
<td>$q_5$</td>
</tr>
<tr>
<td>$q_5$</td>
<td>$q_0$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>$T(q_i,a)$</th>
<th>$T(q_j,a)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$</td>
<td>$q_2$</td>
<td>$q_3$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$q_3$</td>
<td>$q_3$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$q_0$</td>
<td>$q_0$</td>
</tr>
<tr>
<td>$q_4$</td>
<td>$q_0$</td>
<td>$q_0$</td>
</tr>
<tr>
<td>$q_5$</td>
<td>$q_3$</td>
<td>$q_3$</td>
</tr>
</tbody>
</table>

$[\text{Action a}]$

$T(q_1,a) = q_3$

$T(q_5,a) = q_3$

$E[q_3,q_3] = 1$
# FSM Minimization

## Result:

State transition table $T$

<table>
<thead>
<tr>
<th>State</th>
<th>In</th>
<th>Out</th>
<th>Next state</th>
<th>new</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>msg1</td>
<td>-</td>
<td>$q_1$</td>
<td></td>
</tr>
<tr>
<td>$q_0$</td>
<td>msg0</td>
<td>-</td>
<td>$q_2$</td>
<td></td>
</tr>
<tr>
<td>$q_1$</td>
<td>-</td>
<td>ack1</td>
<td>$q_3$</td>
<td>$q_0$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>-</td>
<td>ack0</td>
<td>$q_0$</td>
<td>$q_0$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>msg0</td>
<td>-</td>
<td>$q_4$</td>
<td>$(q_2)$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>msg1</td>
<td>-</td>
<td>$q_5$</td>
<td>$(q_3)$</td>
</tr>
<tr>
<td>$q_4$</td>
<td>-</td>
<td>ack0</td>
<td>$q_0$</td>
<td></td>
</tr>
<tr>
<td>$q_5$</td>
<td>-</td>
<td>ack1</td>
<td>$q_3$</td>
<td></td>
</tr>
</tbody>
</table>

Equivalence table $E$

<table>
<thead>
<tr>
<th>$q_0$</th>
<th>$1$</th>
<th>[]</th>
<th>[]</th>
<th>[]</th>
<th>[]</th>
<th>[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$</td>
<td>$0$</td>
<td>$1$</td>
<td>[]</td>
<td>[]</td>
<td>[]</td>
<td>[]</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$0$</td>
<td>$0$</td>
<td>$1$</td>
<td>[]</td>
<td>[]</td>
<td>[]</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$1$</td>
<td>$0$</td>
<td>$0$</td>
<td>$1$</td>
<td>[]</td>
<td>[]</td>
</tr>
<tr>
<td>$q_4$</td>
<td>$0$</td>
<td>$0$</td>
<td>$1$</td>
<td>$0$</td>
<td>$1$</td>
<td>[]</td>
</tr>
<tr>
<td>$q_5$</td>
<td>$0$</td>
<td>$1$</td>
<td>[]</td>
<td>$0$</td>
<td>$0$</td>
<td>$0$</td>
</tr>
</tbody>
</table>

| $q_0$ | $q_1$ | $q_2$ | $q_3$ | $q_4$ | $q_5$ |
FSM Minimization

Result:

State transition table $T$

<table>
<thead>
<tr>
<th>State</th>
<th>In</th>
<th>Out</th>
<th>Next state</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>msg1</td>
<td>-</td>
<td>$q_1$</td>
<td></td>
</tr>
<tr>
<td>$q_0$</td>
<td>msg0</td>
<td>-</td>
<td>$q_2$</td>
<td></td>
</tr>
<tr>
<td>$q_1$</td>
<td>-</td>
<td>ack1</td>
<td>$q_3$</td>
<td>$q_0$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>-</td>
<td>ack0</td>
<td>$q_0$</td>
<td></td>
</tr>
<tr>
<td>$q_3$</td>
<td>msg0</td>
<td>-</td>
<td>$q_4$</td>
<td>$(q_2)$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>msg1</td>
<td>-</td>
<td>$q_5$</td>
<td>$(q_3)$</td>
</tr>
<tr>
<td>$q_4$</td>
<td>-</td>
<td>ack0</td>
<td>$q_0$</td>
<td></td>
</tr>
<tr>
<td>$q_5$</td>
<td>-</td>
<td>ack1</td>
<td>$q_3$</td>
<td></td>
</tr>
</tbody>
</table>
FSM Composition

Composition of $Q^1$ and $Q^2$

- $Q = Q^1 \times Q^2$, $M' = M^1 \cup M^2$
- $q_0 = q_0^1q_0^2$
- For each state $q^1q^2$ define transitions (non-deterministic):
  $\forall a: T(q^1q^2,a) = T^1(q^1,a) \cup T^2(q^2,a)$
- Minimize the machine
Coupling of FSMs

Synchronous coupling

Sender

Receiver
Synchronous coupling, 2nd example

- Synchronous coupling ignores the transmission delay
Coupling of FSMs

Asynchronous coupling

- Incoming messages are added to the input queue
- The process consumes the first message in queue (FIFO)
Coupling of FSMs

Asynchronous coupling

Terminal 1

Terminal 2

Specification incomplete!

The protocol blocks here, because...

...T2 cannot send ack0 in q2

...T1 cannot send ack1 in q2
Extended FSMs

- Abstract model for communicating processes
- can be transformed into program code
- ... or verification languages (e.g. PROMELA)

- The Specification and Description Language (SDL) is based on Extended FSMs
SDL and MSC

- **Specification and Description Language (SDL)** [ITU-T Recommendation Z.100]
  - originally developed for the specification of telecommunication systems (esp. telephone exchanges)
  - formal language, based on extended FSMs
  - used, e.g., for ISDN protocols, IEEE standards
  - strong tool support

- **Message Sequence Charts (MSC)** [ITU-T Z.120]
  - originally part of SDL; similar to UML sequence diagrams

Source: [http://www.itu.int/ITU-T/studygroups/com10/languages/]
SDL Example

SDL process diagram for Lynch’s protocol

process transceive

get next char

receive

NACK

ACK

ERR

get next char

ACK

NACK

-

next state = previous state

process start

procedure call

state

receive

send

Network Protocol Design and Evaluation
Stefan Rührup, Summer 2009

Computer Networks and Telematics
University of Freiburg
SDL Elements

- SDL describes concurrent processes and their interaction
- Basic concept: Extended (communicating) finite state machines
- Graphical and textual notation
  - SDL/GR (graphic representation)
  - SDL/PR (phrase representation)
- An SDL specification of a system describes
  - Structure
  - Communication
  - Behaviour
  - Data
Basic SDL Elements

- Processes describe *behavior* (Extended FSM)
- They run in parallel and can communicate
- Processes are grouped into blocks
Basic SDL Elements

- Blocks describe the *structure*.
- They can be connected to or contained in other blocks.
- The outermost block is called the system.
- Blocks and processes are called **agents**.

\[<\text{system} >\]

\[<\text{block} >\]

\[<\text{process} >\]

\[<\text{EFSM} >\]

\[<\text{block} >\]

\[<\text{process} >\]

\[<\text{EFSM} >\]
Basic SDL Elements

- Agents communicate
  - asynchronously by a signal (via a channel) or
  - synchronously by a procedure call
- Channels describe the *communication* paths
System: the enclosing block that interfaces the environment

The overall system consists of blocks and processes (agents)

Blocks are structural elements. They can contain other blocks and/or processes.

[R. Reed, SDL-2000 Presentation, sdl-forum.org/sdl2000present/]
Processes

- Processes describe *behavior*
- Processes usually contain an extended finite state machine
- They are *not concurrent*
- They cannot contain blocks
- Processes communicate by signals.
- Processes can contain and/or call procedures

[R. Reed, SDL-2000 Presentation, sdl-forum.org/sdl2000present/]
Definition of a Block
Definition of a Channel

• Channels are used to interconnect agents
• ...also called communication paths or signal routes (distinction between channels and signal routes in SDL-88)
• Signals are sent via channels

channel types:

channel name    signals

ch1

[msg,ack]

bidirectional, without delay
bidirectional, without delay
bidirectional, with delay
unidirectional, with delay
Processes

[Diagram of processes with states and procedure calls]

- Diagram heading
- Start state
- Procedure call
- State
- Connector (for splitting diagrams)
- Procedure (reference to)
- Input
- Appearance of p2 inside a block:
Procedures

![Diagram of a procedure with states and symbols]

- **Diagram heading:** "procedure pr 1(1)"
- **Procedure start:** Indicated by the start symbol
- **Create process:** Symbol representing the creation of a process
- **State:** Diagrams showing the state of the processes
- **Return symbol:** Indicates the return point in the procedure
- **Page number:** "40"
- **Input:** Arrows indicating input connections
Structuring elements

[R. Reed, SDL-2000 Presentation, sdl-forum.org/sdl2000present/]

Network Protocol Design and Evaluation
Stefan Rührup, Summer 2009

Computer Networks and Telematics
University of Freiburg
Describing Behavior: Processes

- Behavior is specified by processes, following the concept of an extended FSM.
- Processes can
  - receive, save, and send signals
  - set and reset timers
  - manipulate variables
  - call procedures
  - create other processes
Elements of a process

- **Start symbol**: (only one per agent)
- **State**: (S1, S2)
  - All states (except those listed)
- **Return**: Previous state
- **Description**: Text extension
- **Task**: c := 0;
- **Procedure Call**: proc
- **Procedure Insertion**: proc (reference)
- **Termination**: X
Branches

(outcome) expr. ELSE decision (branch)

join

connector label

BREAK label parts can be separated by BREAK and connectors
Examples of Decisions

- $x = y$
- $x - y$
- $\text{type} = \text{"A"}$
- $\text{length(header)} = 0$
- $\text{length(header)} = 16$

[ sdl-forum.org/sdl88tutorial/ ]
Processes and signals (1)

- Every process instance has its input queue (FIFO)
- Signals can be received at any time
- Signals from the so-called complete valid input signal set are added to the queue
- If a process is in a certain state and the queue is not empty and there are signals associated with transitions from that state, then the signal is removed from the queue and the transition is triggered.
- For unspecified signal/state combinations, the signal is consumed without any action (implicit transition)

[](http://sdl-forum.org/sdl88tutorial/)
 Processes and signals (2)

- Processes communicate asynchronously via FIFO queues
- Each process has exactly one input queue
I/O Elements (1)

SIGNAL A,B;

SIGALLLIST sl1 = A,B;
SIGALLLIST sl2 = sl1,C;

**signal declaration**

**output signal**

**input signal**

**priority input**

**save signal (msg. remains in queue, no outgoing transition)**

**save all other signals**

**continuous signal with enabling condition**

**abort**
I/O Elements (2)

Sending to a specific receiver:

A TO Dest

Dest: Process ID

A TO SELF

sending a self-message

A TO SENDER

sending back to the sender

A VIA ch1

sending via a channel

Further addresses:

PARENT
the creating instance

OFFSPRING
last created instance by this instance
I/O Elements (3)

Input and Output in layered protocols
(Notation used in IEEE Standards, not official part of Z.100)

*pointer or wedge to the left*

- in_pkt
- out_pkt

Signals from or to processes logically above or parallel to this process

*pointer or wedge to the right*

- out_frame
- in_frame

Signals from or to processes logically below this process
Handling signals, Example (1)

- The process is in state “S1”
- Message “c” is first in queue
Handling signals, Example (2)

- “c” is saved and remains ‘passively’ in queue
Handling signals, Example (3)

- “a” is consumed and removed from the queue
- It triggers the transition to S2
Handling signals, Example (4)

- “c” is now consumed and triggers the transition to S3
Handling signals, Example (5)

- If a transition leads back to the same state, a signal triggering this transition is effectively discarded.
### I/O Notation

Sending to a specific receiver:

- **A TO Dest**
  
  *Dest: Process ID*

- **A TO SELF**
  
  *sending a self-message*

- **A TO SENDER**
  
  *sending back to the sender*

- **A VIA ch1**
  
  *sending via a channel*

Further addresses:

- **PARENT**
  
  *the creating instance*

- **OFFSPRING**
  
  *last created instance by this instance*
Sending signals, Example

- Signal “A” is sent via channel ch2
- “A” is put into the input queue of process p2
Variables

- Variables are declared in a text symbol
- They are manipulated in an task

```
DCL counter Integer := 0,
    increment Integer := 1;

counter := counter + increment;
```

[ sdl-forum.org/sdl88tutorial/ ]
Timers are self-messages which are added to the input queue.
Timers, Example

expirytime := expirytime + period;

SET (expirytime, T)

Wait

T

PeriodAction

DCL expirytime Time := 0.0;
DCL period Duration := 13;
TIMER T;

Alternative:
expirytime := NOW + period;

[ SDL-forum.org/sdl88tutorial/ ]
Passing data variables

- Signals can contain data values
- Input and output must be compatible

Block b1

Process P1

Process P2

[SDL-forum.org/sdl88tutorial/]
Import and Export of Variables

- Instead of passing a signal, a variable can be exported by a process and imported by another process.

```
DCL EXPORTED sum Integer;

EXPORT(sum)

IMPORT(sum, exporterID)
```
Process creation and termination

**Block** Example

```
p1 -> [A] p1
```

**Process** P1

```
S1
A
P2(4,True)
S2
```

**Dynamic process creation**

**Process** P2

```
FPAR V1 Integer, V2 Boolean
```

```
S1
A
```

```
B
```

process terminates

[ sdl-forum.org/sdl88tutorial/ ]
Composite States

[R. Reed, SDL-2000 Presentation, sdl-forum.org/sdl2000present/]
Exceptions

Exception definition

State with associated exception handler

Raising the exception

Exception handler

Handle
Data Types

- SDL follows the concept of Abstract Data Types (ADT)
- ADT = sorts + operators
- Predefined types (with operations):
  - Boolean, Character, Charstring, Integer, Natural, Real, Duration, Time, Bitstring, Octet, Octetstring, Pid
  - Parameterized: Strings (i.e. lists) of any type, Arrays, Structures, Choice, Powerset, Bag
  - Different sets of predef. types in SDL-88 and SDL-2000
- User-defined types:
  Value types, Object types, Syntypes (with range check)

[R. Reed, SDL-2000 Presentation, sdl-forum.org/sdl2000present/]
Data Types, Example

```
object type Linkedlist
    <type Elementsort>
    struct
        prev, next this Linkedlist;
        data Elementsort;
    operators
        "in" (Elementsort, Linkedlist) -> Boolean;
    methods
        delete (Elementsort);
        operator "in" referenced;
        method delete referenced;
endobject type Linkedlist;
object type Natlist
    inherits Linkedlist <Natural>
endobject type Natlist;
dcl primes Natlist
    := (. Null, Null, 1 .);
```

[R. Reed, SDL-2000 Presentation, sdl-forum.org/sdl2000present/]

Object orientation

- Classes and objects in SDL: types and instances
- All instance definitions (agents, states...) define an agent type implicitly
- Explicit definition:

```
block type B

block type B2
  inherits B adding

block type C
  b:B2
  B2
  c
```

SDL/GR vs. SDL/PR

[SDL/GR graphical representation]

system convert
  s
  c_in
  B
  c_out
  t

signal s,t;

SDL/PR phrase representation

system convert;
  signal s,t;
  channel c_out nodelay from B to env with t;
  endchannel c_in;
  channel c_in nodelay from env to B with s;
  endchannel c_out;
  block B referenced;
endsystem convert;

block B;
  channel rin nodelay from env to P with s;
  endchannel rin;
  channel rout nodelay from P to env with t;
  endchannel rout;
  process P referenced;
  connect c_out and rout;
  connect c_in and rin;
endblock B;

process P;
  start;
  nextstate idle;
  state idle;
    input s;
    output t;
    nextstate idle;
endstate idle;
endprocess P;

[R. Reed, SDL-2000 Presentation, sdl-forum.org/sdl2000present/]

Network Protocol Design and Evaluation
Stefan Rührup, Summer 2009

Computer Networks and Telematics
University of Freiburg
SDL in the development process

Scope of SDL+ (SDL and MSC with ASN.1) and recommended methodology

[ITU-T Z.100 Supplement 1]
SDL in practice: 802.11 Specification

System specification (part)

[IEEE Std. 802.11-2007]
SDL in practice: 802.11 Specification

Transmission block specification (part)

[IEEE Std. 802.11-2007]
SDL in practice: 802.11 Specification

Backoff process specification (part)

[IEEE Std. 802.11-2007]
History of SDL

- **1968** ITU-T study on the impact of stored program control (SPC) systems (telephone exchanges)
- **1972** follow-up study on languages for human-machine interaction, specification and description, and programming
- **1976** first SDL standard (CCITT Orange book) with basic graphical language
- **1980** description of semantics (CCITT Yellow book)
- **1984** SDL becomes a formal language (CCITT Red book), data elements, graphical and textual notation

[R. Reed, “Notes on SDL-2000 for the new millennium, Computer Networks (35), 2001]
History of SDL

- **1988** formalization completed, syntax, language grammar and semantics consolidated. SDL-88 is the foundation of all subsequent versions. [sdl-forum.org/sdl88tutorial/]

- **1992** object features introduced in SDL-92

- **1995** SDL with ASN.1 (ITU-T Recommendation Z.105)

- **1996** SDL-96 = SDL-92 + corrections and extensions

- **1999** object modeling and a new data model in SDL-2000

[R. Reed, “Notes on SDL-2000 for the new millennium, Computer Networks (35), 2001”]
SDL and UML History

SDL-76

SDL-88

SDL-92

SDL-2000

SDL UML Profile

MSC-92

Harel’s state charts (’87)

UML draft (1995)

UML 1.1 (1997)

UML 2.0 (2005)

UML 2.2 (2009)

SysML 1.0 (2007)

Booch, OMT

SDL UML Profile

MSC-2000

state machines

SDL-2000

MSC-92

SDL-88

SDL-92

SDL-76

UML 1.1 (1997)

UML 2.0 (2005)

UML 2.2 (2009)

SysML 1.0 (2007)

Booch, OMT

Harel’s state charts (’87)

UML draft (1995)

UML 1.1 (1997)

UML 2.0 (2005)

UML 2.2 (2009)

SysML 1.0 (2007)
# SDL and UML

<table>
<thead>
<tr>
<th>UML</th>
<th>SDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>collection of notations for describing different views of a system, including structure, state machine, interaction, collaboration etc.</td>
<td>formal language focusing on structural and state machine views</td>
</tr>
<tr>
<td>weak semantics with many variation points</td>
<td>interactions are modeled by MSC</td>
</tr>
<tr>
<td>mapping of subsets $\text{UML}<em>{\text{SDL}} \leftrightarrow \text{SDL}</em>{\text{UML}}$ defined in [ITU Z.109]</td>
<td>complete semantics</td>
</tr>
</tbody>
</table>

Mapping subsets of UML and SDL

Lessons learned

‣ Basic finite state machine models are not sufficient to model concurrent and communicating processes such as network protocols.

‣ Therefore extended FSMs with channels and variables were introduced

‣ Processes in SDL are based on this concept

‣ There are similarities to UML state machines. However, SDL has the stronger semantics
Message Sequence Charts

- Similar to UML Sequence Diagrams
- formal graphical language
- defined in [ITU-T Recommendation Z.120]
  Source: http://www.itu.int/ITU-T/studygroups/com10/languages/
- describes behavior of communicating instances for specific executions (scenarios, traces)
MSC Basics: Instances

**msc cs_operation**

- **client**
- **server**

**time**

**request(sid)**

**message**
- name
- parameters

**response(data,status)**

**input**

**output**

**instance**
- instance name
- instance head
- instance axis
- instance end
MSC Basics: Messages

message from the environment

transmission delay

lost message with intended target

self-message

found message with supposed source

Network Protocol Design and Evaluation
Stefan Rührup, Summer 2009

Computer Networks and Telematics
University of Freiburg
MSC Basic assumptions

- Communication is performed by means of messages
- Sending and receiving is asynchronous
- No event ordering
- There is a global clock
- Events of different instances are ordered via messages (send before receive, partial ordering)
Instance creation and termination

```
(msc cs_operation

client

parameters

(p1, p2)

server

confirm

createline symbol

confirm

stop

stop symbol

Network Protocol Design and Evaluation
Stefan Rührup, Summer 2009

Computer Networks and Telematics
University of Freiburg
```
Timers

- **Timer start**
- **Timeout**
- **Timer stop**

**timer name** (required), **timer instance name** (optional)

- `t1`
- `t2`, `ti1`
- `t2 [10,15]`
- `t2 [2]`

**duration** (optional) with duration \([\text{min}, \text{max}]\)

- **Restart**
- **Timeout**

**duration** = \([0, \infty]\) unless otherwise specified
Network Protocol Design and Evaluation
Stefan Rührup, Summer 2009

Computer Networks and Telematics
University of Freiburg

Conditions

```
msc cs_operation

client

when connected

req

response

process request

server

disconnected

```

Guarding condition

Setting condition
Conditions

- Conditions contain labels (condition names)

- **Setting conditions**
  - a state-like condition requires setting of the respective labels associated with the covered instances

- **Guard conditions**
  - true, if the labels have a non-empty intersection with the labels associated with the covered instances
  - may contain boolean expressions
  - dynamic variables of the guard are only from the active instance (only one instance can be active)
MSC Reference

**MSC cs_operation**

- Client
- Dispatcher
- Server

**MSC ds_operation**

- Dispatcher
- Server

**MSC reference**

- MSC reference name

**Gate**

- g1
- g2

**Messages**

- req
- response
- conn
- reply(state)
- ack
- data
- response
Network Protocol Design and Evaluation
Stefan Rührup, Summer 2009

MSC Reference

```
msc cs_operation

client       dispatcher       server

req

ack

g1 ← gate

conn_establishment ← g2

gate

loop<2,4>

ds_operation

disconnect

fin

MSC reference
MSC reference name
executed 2 - 4 times

further execution options:
- **exc** exception
- **opt** optional
- **par** parallel
- **seq** sequential
```
MSC Reference Example

[Grabowski, Reed: “ASN.1, MSC, SDL and TTCN Today”, Tutorial, WITUL 2004]
Inline expressions

```
msc example

Inst1

conn_req

conn_ack

t1

data

Inst2

data_ack

conn_reset

alt

t1

separator

Two alternatives

Inline expression

91
```
Nested and guarded inline expressions

```
\textbf{msc} \texttt{inline\_example\_2}

\begin{verbatim}
\textbf{Inst1} \quad \textbf{Inst2}

\textbf{when} \ \texttt{conn\_established}

\textbf{loop}\langle 0,\infty \rangle

\textbf{alt}

\textbf{guarded inline expressions}

\textbf{Guard}

\textbf{Nested inline expression}
\end{verbatim}
```
Time observation

**Time constraints**

```
msc time_example

Inst1

msg1 time &t

Inst2

call proc1(x)

msg2 time (0,2*t)

Inst3

disconnect
```

Method calls and control flow

```
msc call_example

Inst1  Inst2  Inst3

--call p1--

--call p2(x,y)--

--call p3--

p1

p2(x,y)

p3

Method call
Method call gate
Method area
Suspension area
Reply
Reply gate

Network Protocol Design and Evaluation
Stefan Rührup, Summer 2009

Computer Networks and Telematics
University of Freiburg
```
High level MSCs

High-level MSCs

- HMSCs describe the combination of basic MSCs
- Elements are references to MSCs and their connections
- HMSCs give an overview of alternative message sequences
- Higher level of abstraction: instances and interactions (message transmissions) are hidden
MSC in practice: 802.11 Specification

Network Protocol Design and Evaluation
Stefan Rührup, Summer 2009
MSC Review

- Graphical formal language for describing inter-object behaviour
- Application: specifying requirements in the form of *scenarios*, documenting test cases etc.
- Partial order semantics, no causality
- Extensions: High level MSCs
- Are MSCs sufficient to generate code?
Shortcomings of MSC Semantics

- Existential or universal?
  - Description of a sample run or mandatory protocol?
- Safety and Liveness
  - MSCs only express safety (no more bad things happen), but not liveness (something will eventually happen)
  - Progress cannot be enforced
- No simultaneous events
- Rudimentary timing and conditions without semantics before MSC-2000

Code generation from scenario-based specifications

- Can a state machine be derived from an MSC?
- MSC and semantic variation:
  - Does a system contain at most, at least, or all components specified in the MSC?
  - Is the described message exchange complete? ... or are there other message sequences allowed?

[I. Krüger, R. Grosu, P. Scholz, and M. Broy, “From MSCs to Statecharts”, DIPES’98]
A Semantic Model for MSCs

- A system consists of components and channels
- Components operate by reading input, calculating the output and writing output
- There is a global discrete clock
- Asynchronous communication
- Exact description: The message sequences occur only once and there are no other possible interactions

[I. Krüger, R. Grosu, P. Scholz, and M. Broy, “From MSCs to Statecharts”, DIPES’98]
From an MSC to a Statechart

- 5-stage process:
  1. Projection of MSCs onto the component
  2. Normalization
  3. Transformation into an MSC automaton
  4. Transformation into an automaton
  5. Optimization (minimization etc.)

[I. Krüger, R. Grosu, P. Scholz, and M. Broy, “From MSCs to Statecharts”, DIPES’98]
Example

- Car locking system
- Informal description:
  - Components: KeyControl, left door motor and right door motor
  - The driver can lock or unlock the door with his remote control (signals: “lock” and “unlock”)
  - The locked/unlocked status is set after the motors finished their action and sent a “ready” message

[I. Krüger, R. Grosu, P. Scholz, and M. Broy, “From MSCs to Statecharts”, DIPES’98]
Example MSCs

[I. Krüger, R. Grosu, P. Scholz, and M. Broy, “From MSCs to Statecharts”, DIPES’98]
Focus on the component C of which the state machine should be derived

Remove all other instance axes and all messages that are neither sent nor received by C

[I. Krüger, R. Grosu, P. Scholz, and M. Broy, “From MSCs to Statecharts”, DIPES'98]
Projection

[MSC Diagrams]

[I. Krüger, R. Grosu, P. Scholz, and M. Broy, “From MSCs to Statecharts”, DIPES’98]
Normalization

- A normalized MSC begins and ends with a condition symbol.
- It has exactly these two conditions and a (empty or non-empty) sequence of messages in between.
- MSCs with more condition symbols are split.

(the example MSC is already normalized)

[I. Krüger, R. Grosu, P. Scholz, and M. Broy, “From MSCs to Statecharts”, DIPES’98]
Transformation into MSC Automaton

MSC lockSeq

KeyControl

unlocked

lock

lm_down

rm_down

lm_ready rm_ready

locked

[KeyControl

unlocked

lock

lm_down

rm_down

lm_ready rm_ready

locked]

MSC unlockSeq

KeyControl

locked

unlock

lm_up

rm_up

lm_ready rm_ready

unlocked

[KeyControl

locked

unlock

lm_up

rm_up

lm_ready rm_ready

unlocked]

States = conditions
Transitions = MSCs beginning and ending with the respective conditions

[I. Krüger, R. Grosu, P. Scholz, and M. Broy, “From MSCs to Statecharts”, DIPES’98]
Transformation into CFSM

[I. Krüger, R. Grosu, P. Scholz, and M. Broy, “From MSCs to Statecharts”, DIPES’98]
[I. Krüger, R. Grosu, P. Scholz, and M. Broy, “From MSCs to Statecharts”, DIPES’98]
Life Sequence Charts

- Extending MSCs by liveness annotations
  (Extension of MSC-96)
- Mandatory (hot) and provisional (cold) elements
- Existential and universal charts
- Asynchronous and instantaneous messages
- Conditions and invariants

[W. Damm, D. Harel: “LSCs: Breathing Life into Message Sequence Charts”,
LSC Basics (1)

LSC: Example1a
AC: Act
AM: Invariant

Env | Inst1 | Inst2 | Inst3
---|---|---|---

| | | | |
| | | | |
| async1 | sync2 | async2 | sync2r |
| | | | |
| condition1 | sync3 | | |
| | | | |
| | | | |
| | | | |

[Sync messages: sync1, sync1r, sync2, sync2r, sync3, sync3r]

[Activation modes: initial, invariant, iterative]

[Activation condition]

[Asynchronous message]

LSC Basics (2)

## Safety and Liveness in LSCs

<table>
<thead>
<tr>
<th></th>
<th>Mandatory</th>
<th>Provisional</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Chart</strong></td>
<td>universal</td>
<td>existential</td>
</tr>
<tr>
<td></td>
<td>must be fulfilled by all runs</td>
<td>describes a possible run</td>
</tr>
<tr>
<td><strong>Locations</strong></td>
<td>hot</td>
<td>cold</td>
</tr>
<tr>
<td></td>
<td>progress is enforced</td>
<td>staying without progress is allowed</td>
</tr>
<tr>
<td><strong>Messages</strong></td>
<td>hot</td>
<td>cold</td>
</tr>
<tr>
<td></td>
<td>has to be delivered</td>
<td>may be lost</td>
</tr>
<tr>
<td><strong>Conditions</strong></td>
<td>hot</td>
<td>cold</td>
</tr>
<tr>
<td></td>
<td>must hold, otherwise abort</td>
<td>exit current chart if not met</td>
</tr>
</tbody>
</table>

Timing Constraints

LSC: Example2
AC: Act
AM: Invariant

[m. brill, w. damm, j. klose, b. westphal, h. wittke: live sequence charts: an introduction to lines, arrows, and strange boxes in the context of formal verification. softspez final report 2004]
From LSCs to Symbolic Automata

- Transformation of a LSC into a symbolic timed automaton that describes valid message sequences (*unwinding*)
- Basis for formal verification and validation
- Automaton model: Symbolic timed automaton
  - accepts infinite words (based on Büchi automata)
  - timed words: time is associated to symbols of a word, required that time is non-decreasing and progressing
Unwinding (1)

- Process the elements of the LSC from top to bottom while obeying the partial order
- Elements are *atoms* (instance heads, instance ends, sending and receiving messages, conditions) and the borderline between processed and unprocessed elements is called a *cut*.
- An atom can be enabled and processed
  - if its predecessors on the same axis have been processed
  - in case of a receive element: if the sending operation has been processed
  - in case of a shared condition: if all other condition atoms are enabled
Moving a cut from top to bottom through the LSC:

\[
\begin{array}{cccc}
\text{Inst1} & \text{Inst2} & \text{Inst3} & \text{Inst4} \\
\hline
\text{Cut} & \quad & \quad & \quad \\
\text{Cold cut} & \quad & \quad & \quad \\
\end{array}
\]

LSC: Example 2
AC: Act
AM: Invariant
Unwinding (3)

- Cuts become states in the automaton
- Transitions represent the successor relation among the cuts (successor relation reflects the partial order)
- Cold cuts become the acceptance states
Example (1)

LSC: Example2
AC: Act
AM: Invariant

Inst1  Inst2  Inst3  Inst4

Cut

Cold cut

C0  C1  C2  C3  C4  C5

a  b  c  d  e

q0 → ¬a ∧ ¬b
q1 → a ∧ b
q2 → ¬c ∧ ¬d
q3 → ¬c
q4 → c
q5 → true

Cut

Cold cut
Example (2)

- Language of the automaton describes valid message sequences: \{(a,b,d,c,e), (a,b,c,d,e),...\}

- Note, that this is not a state machine as an implementation model
Code generation and Verification

Requirements

LSC, MSC

descriptive view

State machine(s)

transformation

constructive view

code generation

Implementation
Code generation and Verification

- **Descriptive View**: LSC → Symbolic Automaton
  - Transformation

- **Constructive View**: Requirements → State machine(s)
  - Manual design
  - Code generation
  - Verification/Consistency check
  - Implementation
MSCs, LSCs, and UML

- **MSC-92**
- **MSC-96**
- **MSC-2000**
- **UML 1.1 (1997)**
- **UML 2.0 (2005)**
- **LSC**

Timeline:
- 1990
- 2000
- 2010
Lessons learned

- MSCs are widely accepted as an intuitive way for describing scenarios
- The transformation into a state machine requires an additional semantic model
- LSCs extend MSCs by new elements and a stronger semantic.
- No additional assumptions needed to transform LSCs into state machines
- State machines generated from MSCs/LSCs can specify a single process or define valid message sequences