



ALBERT-LUDWIGS-
UNIVERSITÄT FREIBURG

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

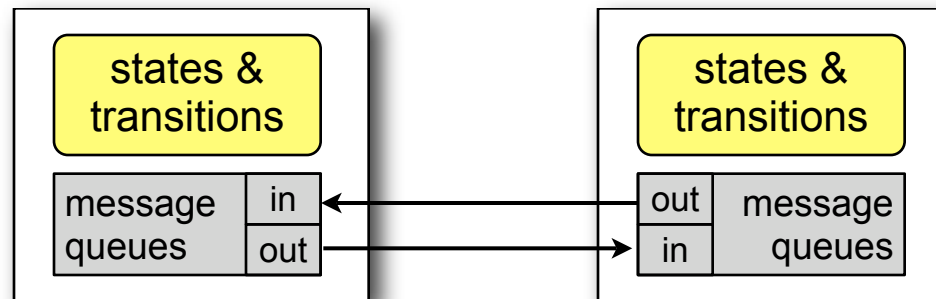
- ▶ **Part II:**
 - Formal state machine models revisited
 - SDL - The specification and description language
 - Describing scenarios with Message Sequence Charts

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

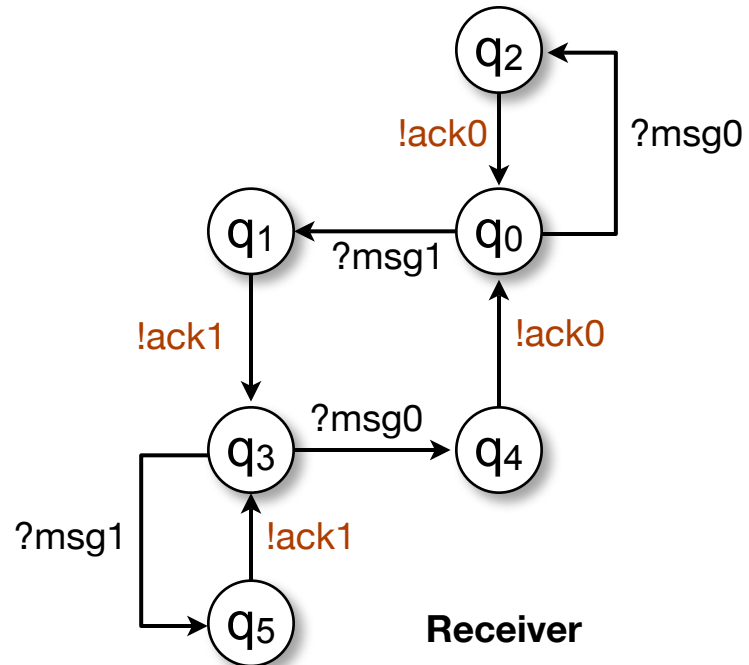
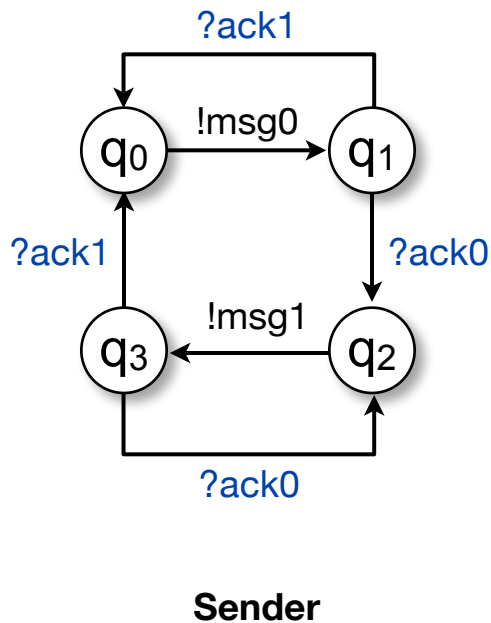
[Holzmann 1991]

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

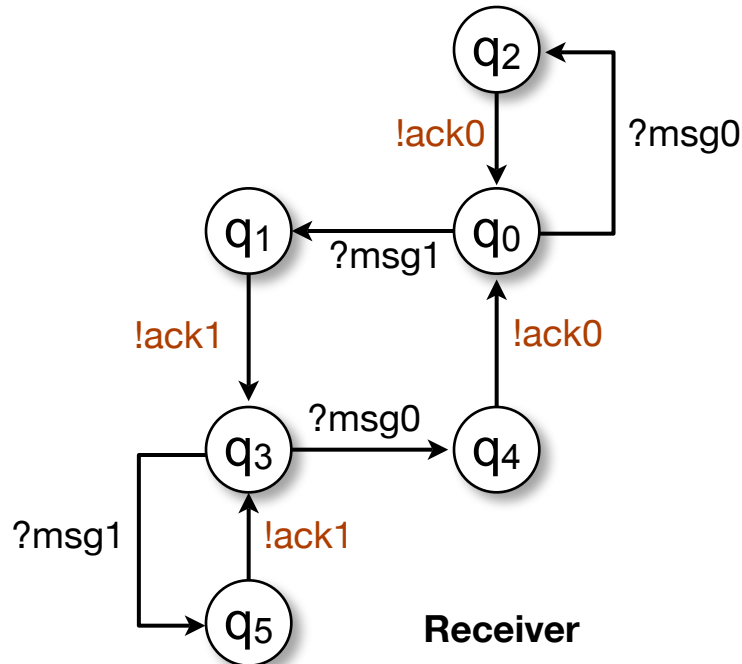
Example for Communicating FSMs (2)

- ▶ CFSM specification of the protocol [Holzmann 1991]



Example for Communicating FSMs (3)

- ▶ State transition table of the receiver protocol [Holzmann 1991]



State	In	Out	Next state
q0	msg1	-	q1
q0	msg0	-	q2
q1	-	ack1	q3
q2	-	ack0	q0
q3	msg0	-	q4
q3	msg1	-	q5
q4	-	ack0	q0
q5	-	ack1	q3

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 ϵ)

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: $T(q_0, !msg) = q_1$
- ▶ 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

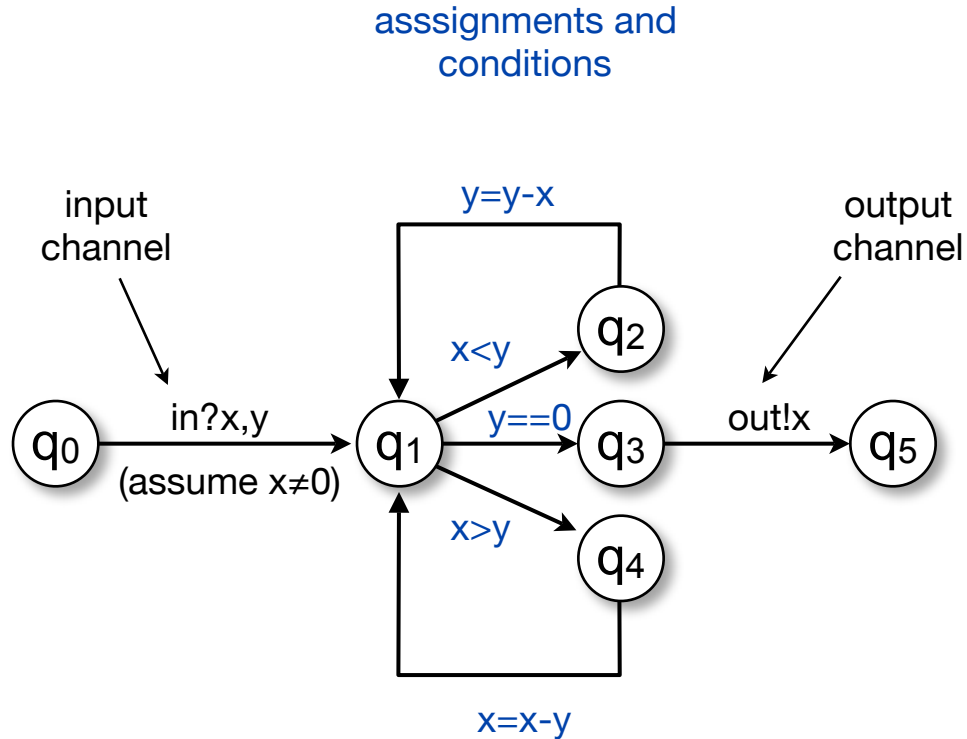
- ▶ **Extensions to the CFSM model:**
 - Variables (integer, finite range)
 - Queues can transfer integer values
 - Set of arithmetic and logical operators

Definition of an Extended FSM

[Holzmann 1991]

- ▶ A *extended finite state machine* is a tuple $(Q, q_0, M, \mathbf{V}, T)$, 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,
 - \mathbf{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 ϵ)

Extended FSMs, Example



[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 $|Q| \times |Q|$, $\text{init} = \text{false}$

for all entries $E[i,j]$

if the states i and j are defined for the same actions, set
 $E[i,j] := \text{true}$ (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 $E[i,j] := \text{false}$

end for

until the number of false entries is not increased

FSM Minimization

Step 1: for all entries $E[i,j]$

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

State transition table T

State	In	Out	Next state
q ₀	msg1	-	q ₁
q ₀	msg0	-	q ₂
q ₁	-	ack1	q ₃
q ₂	-	ack0	q ₀
q ₃	msg0	-	q ₄
q ₃	msg1	-	q ₅
q ₄	-	ack0	q ₀
q ₅	-	ack1	q ₃

Equivalence table E

q ₀	1					
q ₁	0	1				
q ₂	0	0	1			
q ₃	1	0	0	1		
q ₄	0	0	1	0	1	
q ₅	0	1	0	0	0	1
	q ₀	q ₁	q ₂	q ₃	q ₄	q ₅

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 $E[i,j] := \text{false}$

State transition table T

State	In	Out	Next state
q ₀	msg1	-	q ₁
q ₀	msg0	-	q ₂
q ₁	-	ack1	q ₃
q ₂	-	ack0	q ₀
q ₃	msg0	-	q ₄
q ₃	msg1	-	q ₅
q ₄	-	ack0	q ₀
q ₅	-	ack1	q ₃

Equivalence table

		Action a		T(q ₀ ,a)		T(q ₃ ,a)	
		msg1		q ₁	q ₅		
				$E[q_1, q_5] = 1$			
		msg0		q ₂	q ₄		
				$E[q_2, q_4] = 1$			
q ₀	1						
q ₁	0	1					
q ₂	0						
q ₃	1	0	0	1			
q ₄	0	0	1	0	1		
q ₅	0	1	0	0	0	1	
	q ₀	q ₁	q ₂	q ₃	q ₄	q ₅	

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 $E[i,j] := \text{false}$

State transition table T

State	In	Out	Next state
q ₀	msg1	-	q ₁
q ₀	msg0	-	q ₂
q ₁	-	ack1	q ₃
q ₂	-	ack0	q ₀
q ₃	msg0	-	q ₄
q ₃	msg1	-	q ₅
q ₄	-	ack0	q ₀
q ₅	-	ack1	q ₃

Equivalence table E

	q ₀	q ₁	q ₂	q ₃	q ₄	q ₅
q ₀	1					
q ₁	0	1				
q ₂	0	0	1			
q ₃	1	0	0	1		
q ₄	0	0	1	0	1	
q ₅	0	1	0	0	0	1

Action a	T(q ₂ ,a)	T(q ₄ ,a)
ack1	q ₀	q ₀

$E[q_0, q_0] = 1$

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 $E[i,j] := \text{false}$

State transition table T

State	In	Out	Next state
q ₀	msg1	-	q ₁
q ₀	msg0	-	q ₂
q ₁	-	ack1	q ₃
q ₂	-	ack0	q ₀
q ₃	msg0	-	q ₄
q ₃	msg1	-	q ₅
q ₄	-	ack0	q ₀
q ₅	-	ack1	q ₃

Equivalence table E

	q ₀	q ₁	q ₂	q ₃	q ₄	q ₅
q ₀	1					
q ₁	0	1				
q ₂	0	0	1			
q ₃	1	0	0	1		
q ₄	0	0	1	0	1	
q ₅	0	1	0	0	0	1
	q ₀	q ₁	q ₂	q ₃	q ₄	q ₅

Action a

Action a	T(q ₁ ,a)	T(q ₅ ,a)
ack1	q ₃	q ₃

$E[q_3, q_3] = 1$

FSM Minimization

Result:

State transition table T

State	In	Out	Next state	new
q ₀	msg1	-	q ₁	
q ₀	msg0	-	q ₂	
q ₁	-	ack1	q ₃	q ₀
q ₂	-	ack0	q ₀	q ₀
q ₃	msg0	-	q ₄	(q ₂)
q ₃	msg1	-	q ₅	(q ₃)
q ₄	-	ack0	q ₀	
q ₅	-	ack1	q ₃	

Equivalence table E

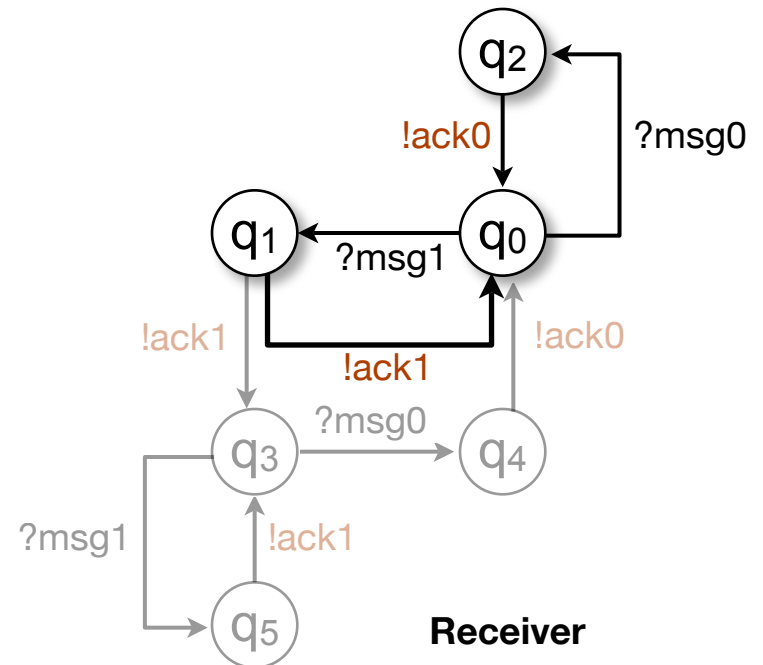
q ₀	1					
q ₁	0	1				
q ₂	0	0	1			
q ₃	1	0	0	1		
q ₄	0	0	1	0	1	
q ₅	0	1	0	0	0	1
	q ₀	q ₁	q ₂	q ₃	q ₄	q ₅

FSM Minimization

Result:

State transition table T

State	In	Out	Next state	new
q ₀	msg1	-	q ₁	
q ₀	msg0	-	q ₂	
q₁	-	ack1	q₃	q₀
q ₂	-	ack0	q ₀	
q ₃	msg0	-	q ₄	(q ₂)
q ₃	msg1	-	q ₅	(q ₃)
q ₄	-	ack0	q ₀	
q ₅	-	ack1	q ₃	

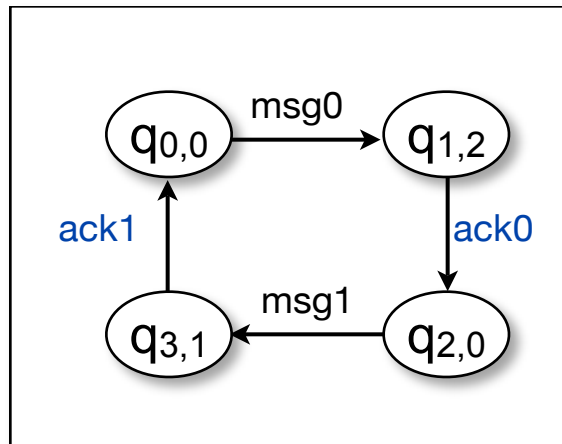
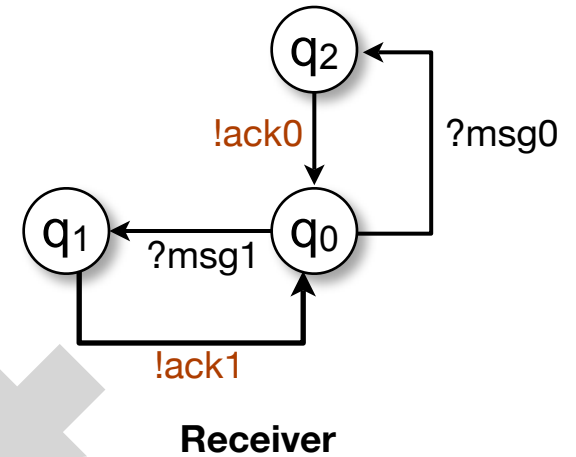
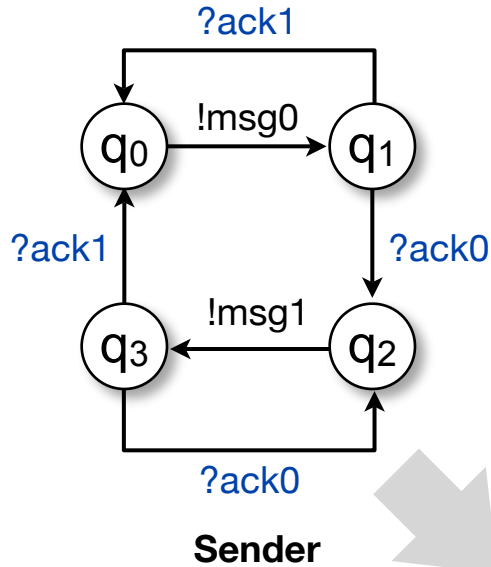


FSM Composition

- ▶ Composition of Q^1 and Q^2
 - $Q = Q^1 \times Q^2, M' = M^1 \cup M^2$
 - $q_0 = q_0^1 q_0^2$
 - Foreach state $q^1 q^2$ define transitions (non-deterministic):
 $\forall a: T(q^1 q^2, a) = T^1(q^1, a) \cup T^2(q^2, a)$
 - Minimize the machine

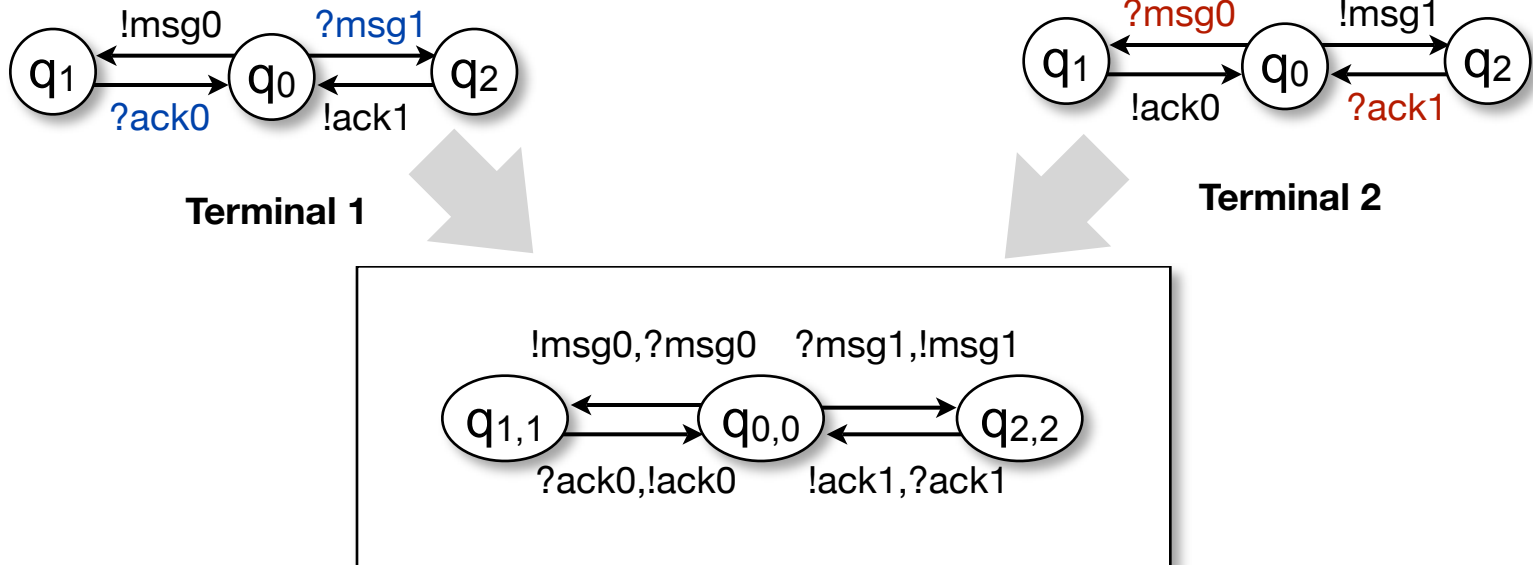
Coupling of FSMs

Synchronous coupling



Coupling of FSMs

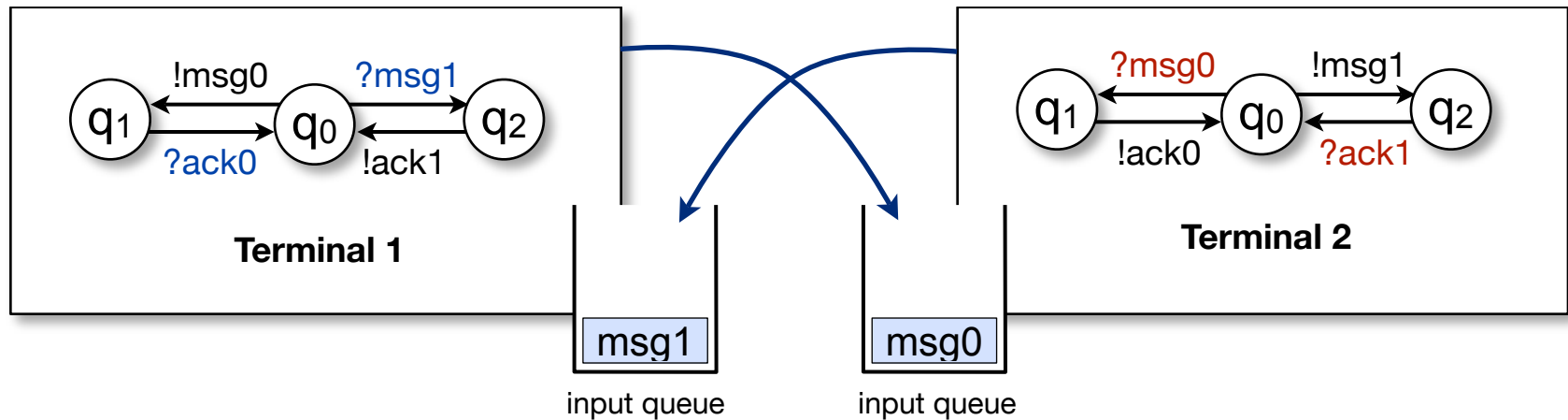
Synchronous coupling,
2nd example



- ▶ Synchronous coupling ignores the transmission delay

Coupling of FSMs

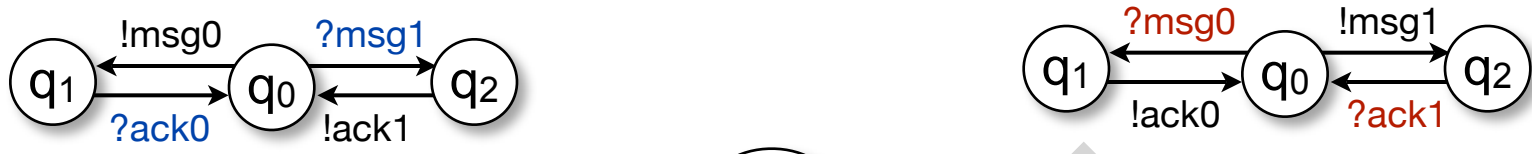
Asynchronous coupling



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

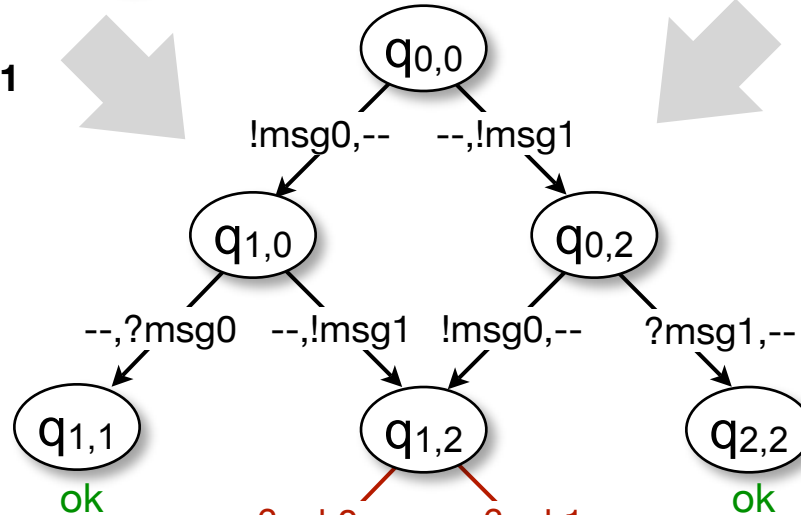
Coupling of FSMs

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

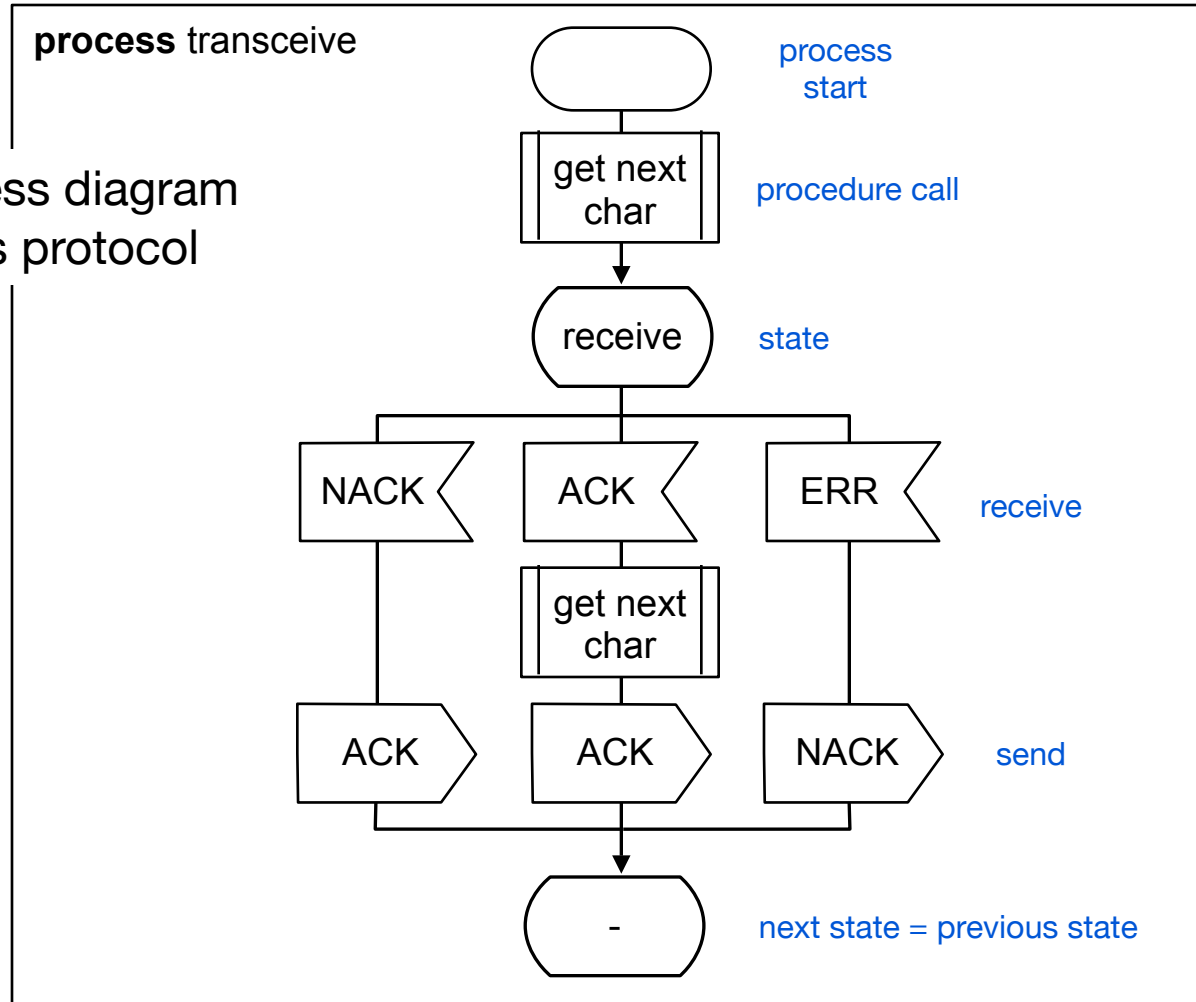
▶ **Message Sequence Charts (MSC)** [ITU-T Z.120]

- 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

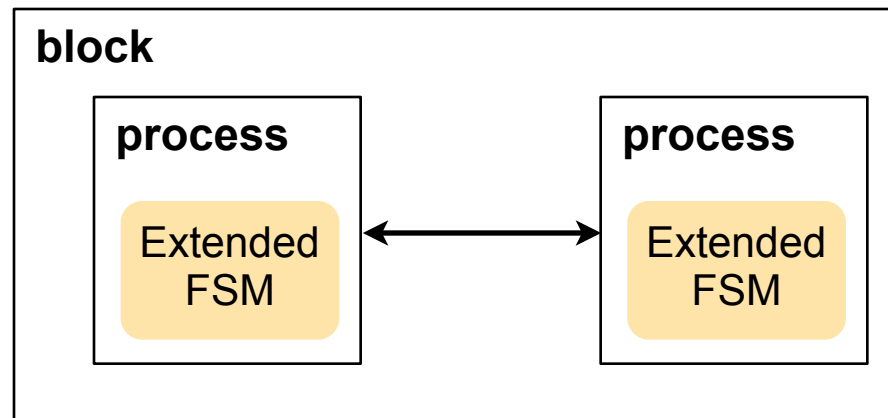


SDL Elements

- ▶ 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

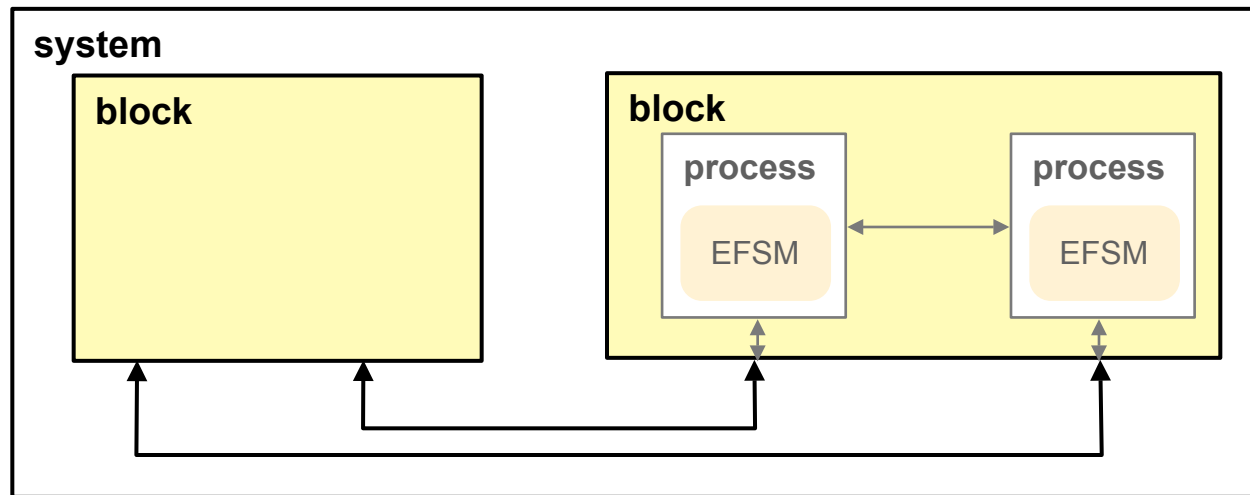
Basic SDL Elements

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



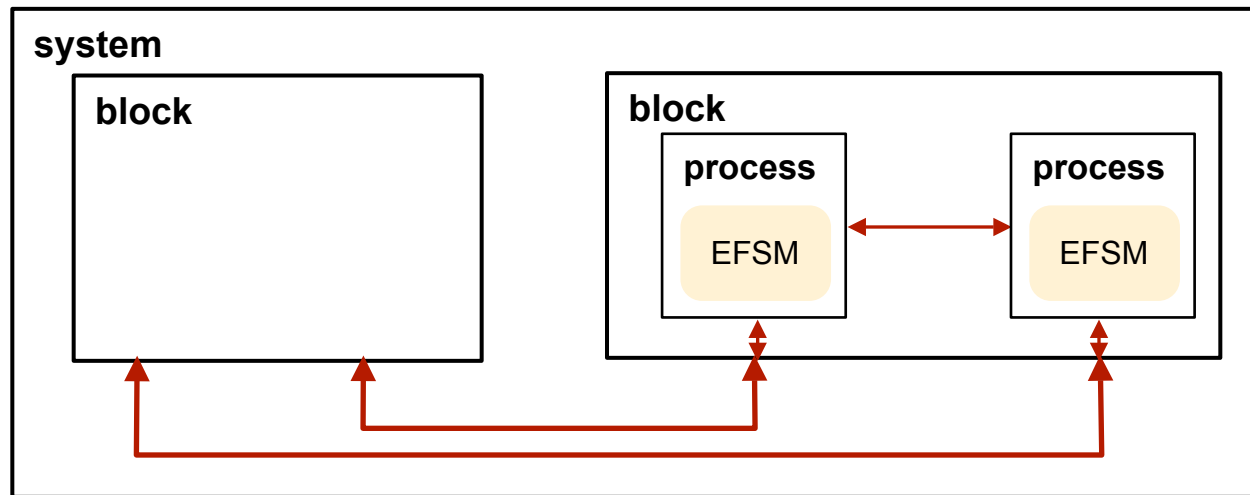
Basic SDL Elements

- ▶ 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**



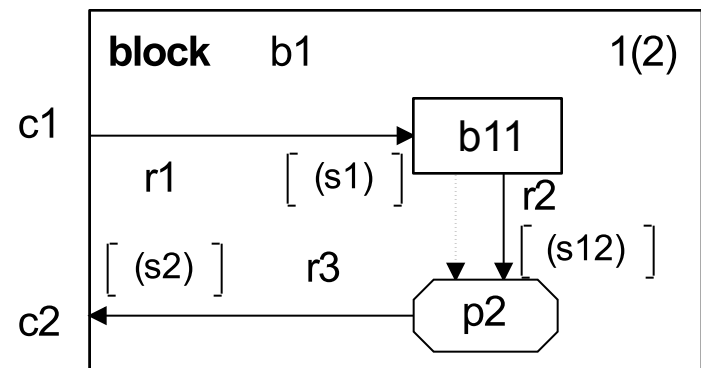
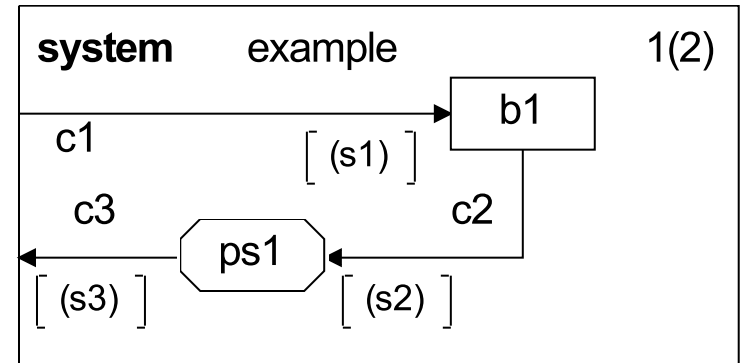
Basic SDL Elements

- ▶ Agents communicate
 - asynchronously by a signal (via a channel) or
 - synchronously by a procedure call
- ▶ Channels describe the *communication* paths



Blocks

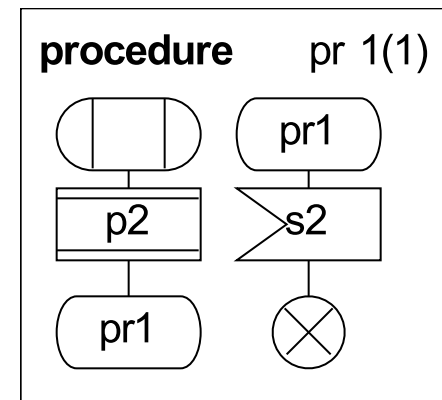
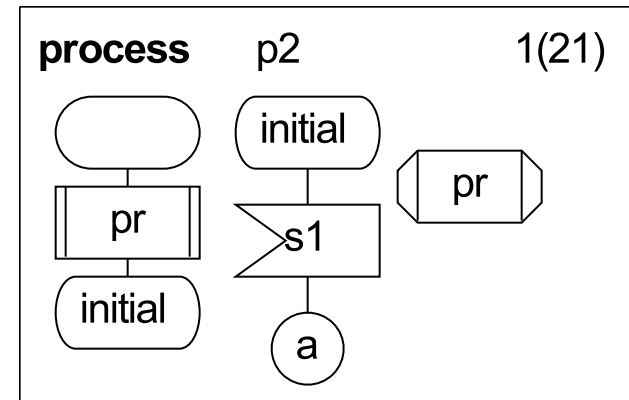
- ▶ System: the enclosing block that interfaces the environment
- ▶ The overall **system** consists of **blocks** and **processes** (agents)
- ▶ Blocks are *structural* elements. They can contain other blocks and/or processes



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

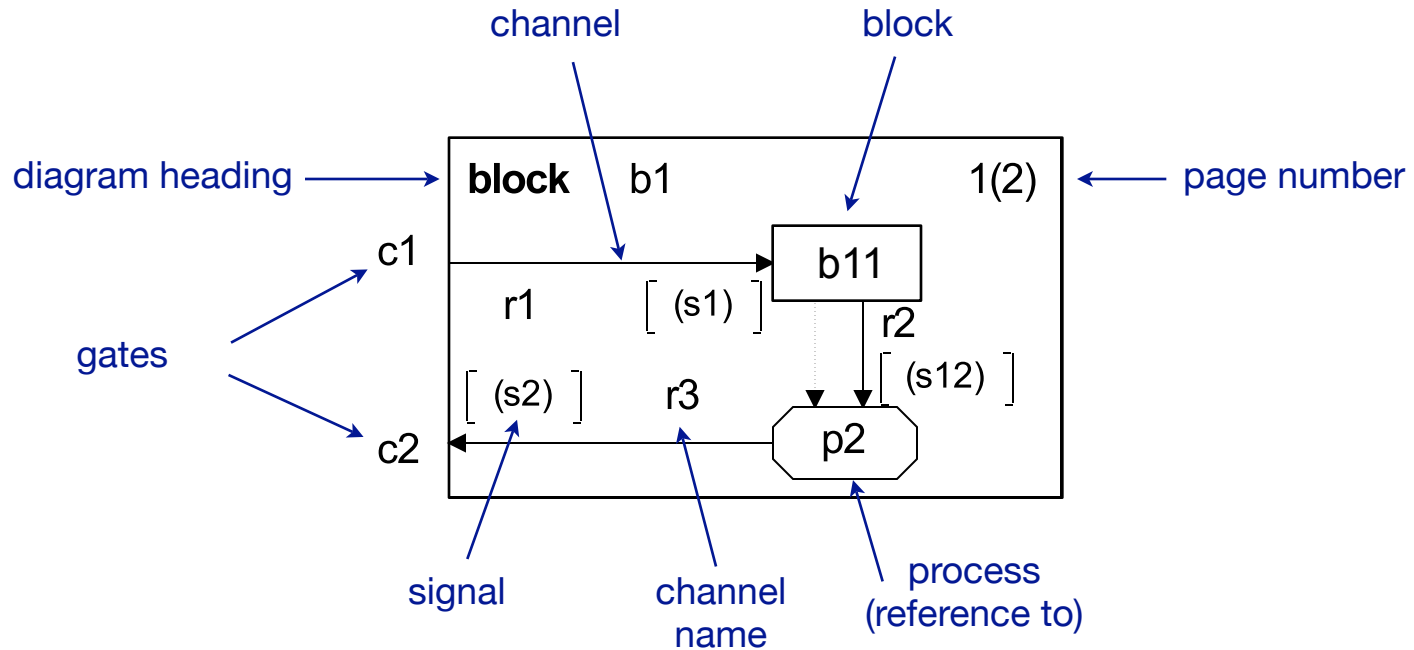
Processes

- ▶ 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



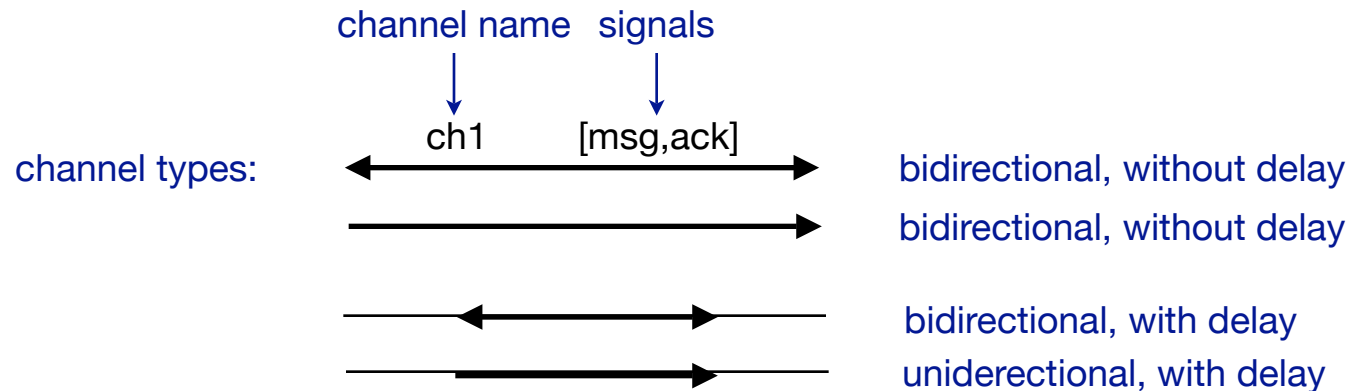
[R. Reed, SDL-2000 Presentation, sdl-forum.org/sdl2000present/]

Definition of a Block

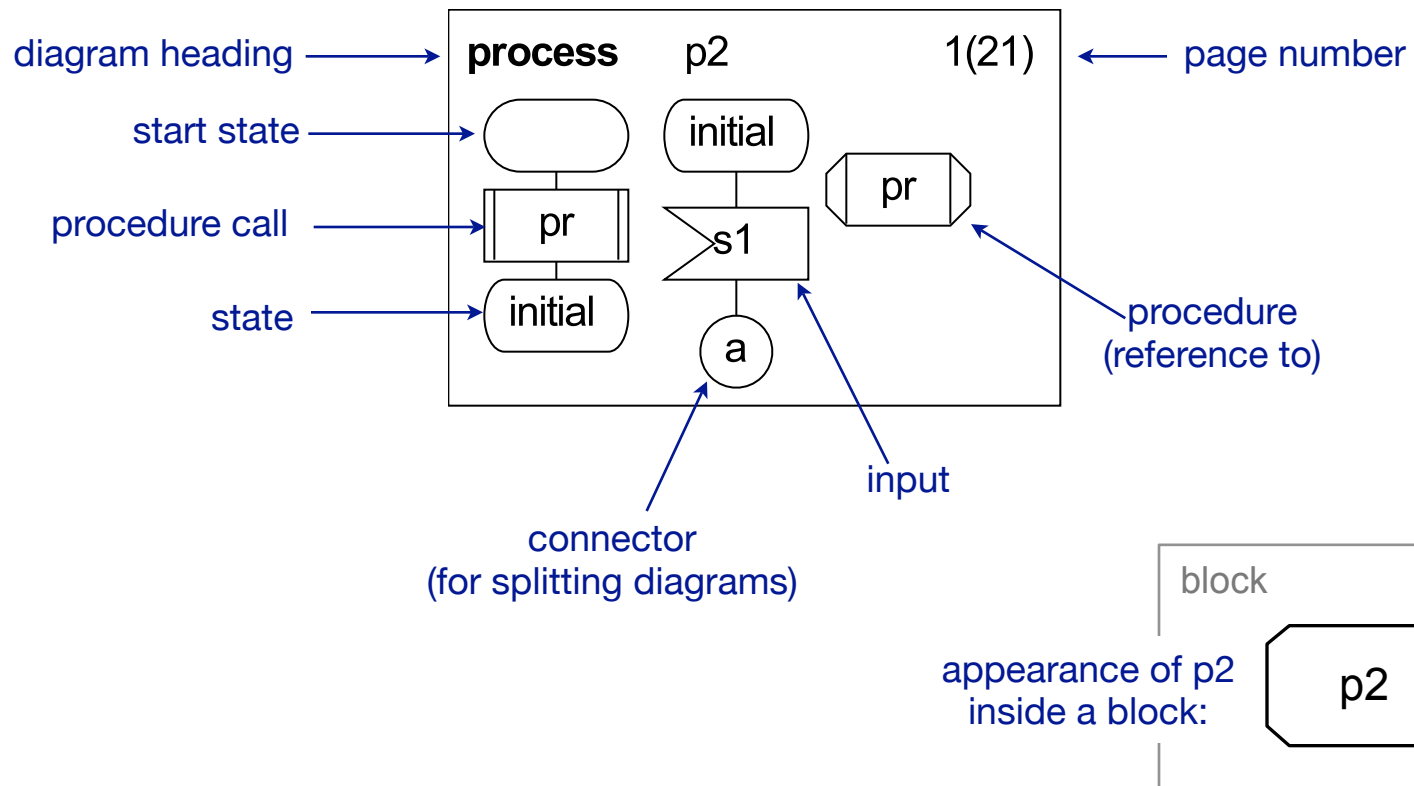


Definition of a Channel

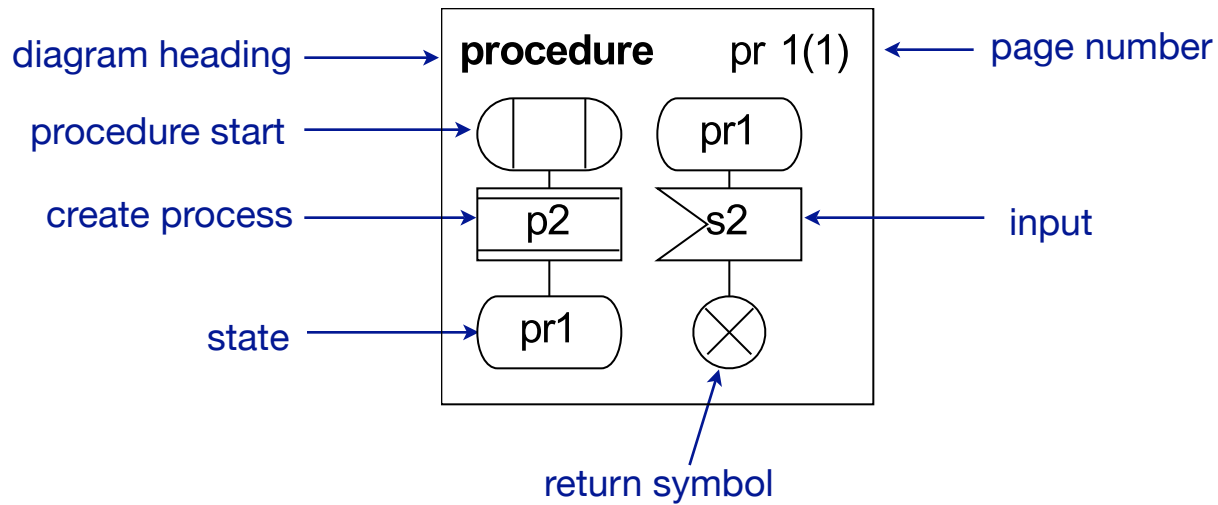
- Channels are used to interconnect agents
- ...also called communication paths or signal routes
(distinction between channels and signal routes in SDL-88)
- Signals are sent via channels



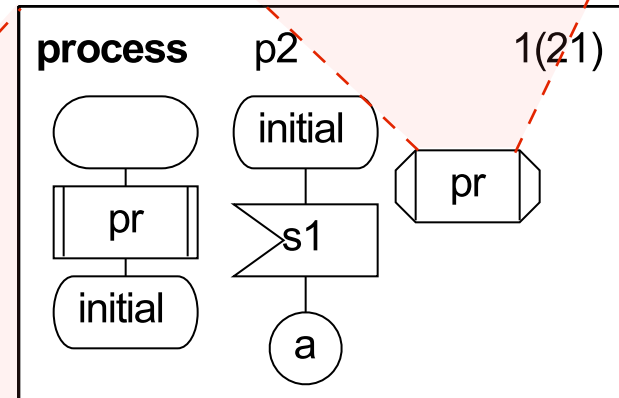
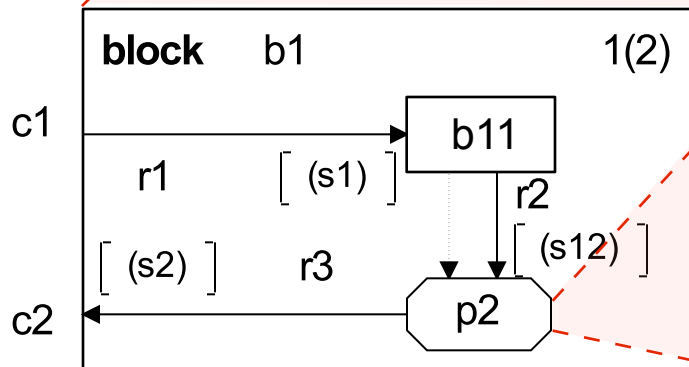
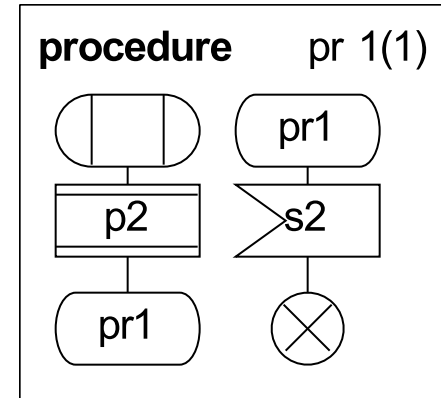
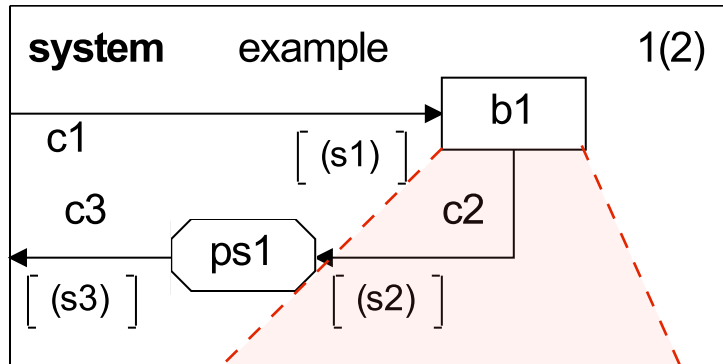
Processes



Procedures



Structuring elements


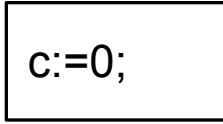

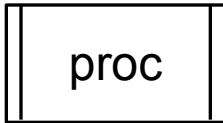
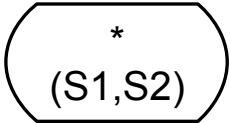



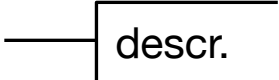


[R. Reed, SDL-2000 Presentation, sdl-forum.org/sdl2000present/]

Describing Behavior: Processes

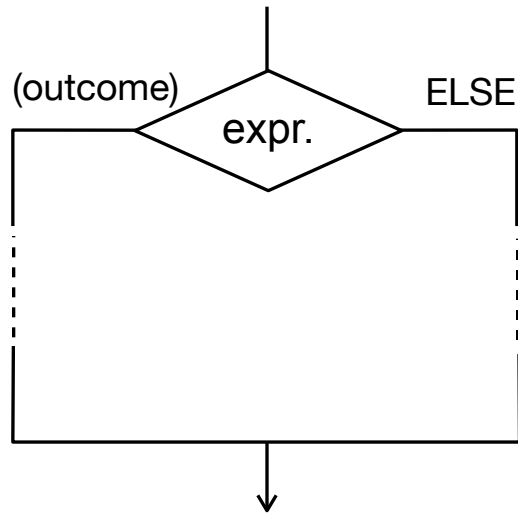
- ▶ Behavior is specified by processes, following the concept of an extended FSM.
- ▶ Processes can
 - receive, save, and send signals
 - set and reset timers
 - manipulate variables
 - call procedures
 - create other processes

Elements of a process

	start symbol (only one per agent)		task
	state		procedure call
	all states (except those listed)		procedure insertion (reference)
	return to previous state		termination
	text extension		

[sdl-forum.org/sdl88tutorial/]

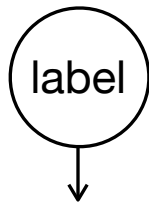
Branches



decision
(branch)

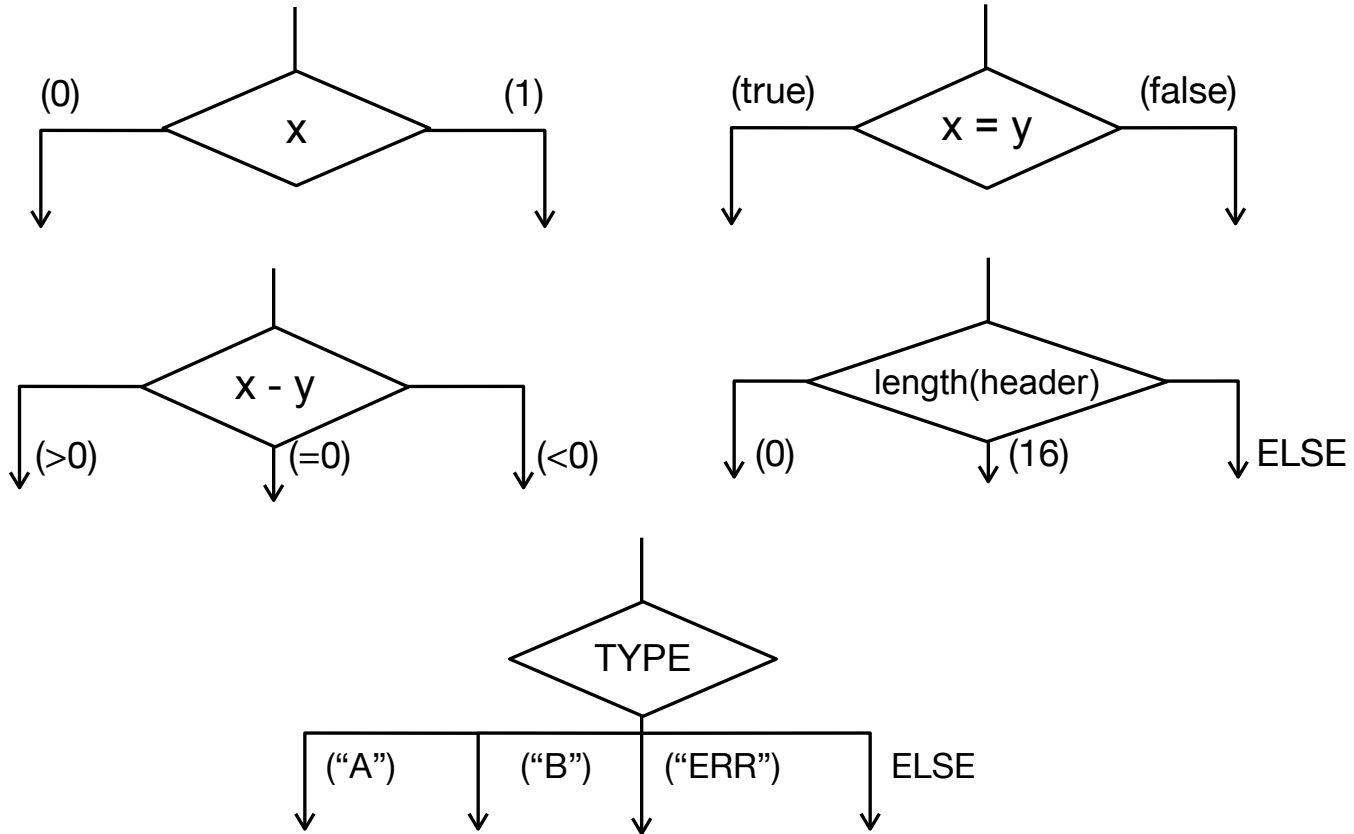
join

connector



parts can be separated
by BREAK and
connectors

Examples of Decisions



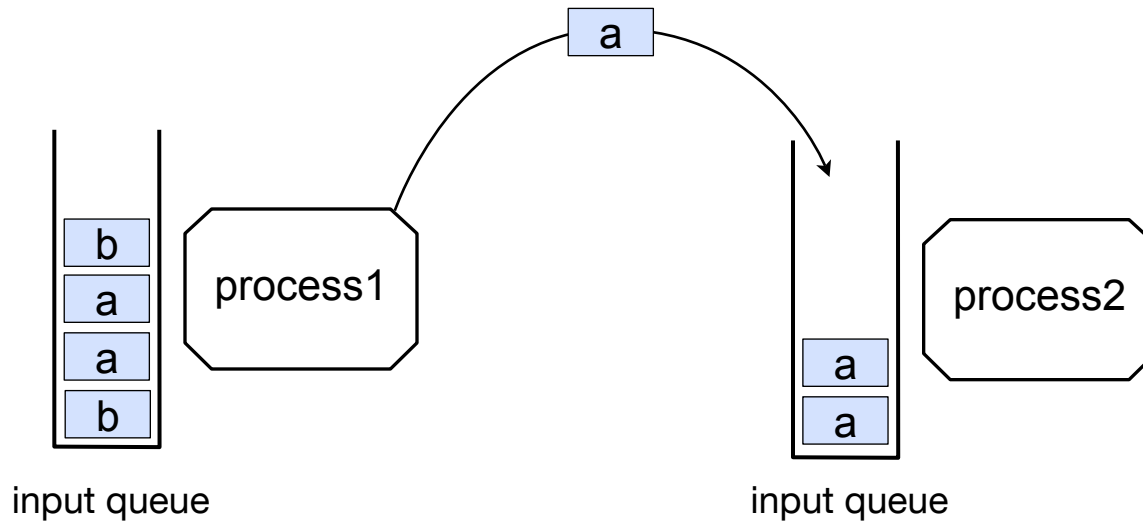
[sdl-forum.org/sdl88tutorial/]

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*)

[sdl-forum.org/sdl88tutorial/]

Processes and signals (2)



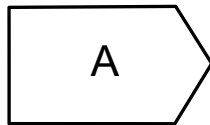
- ▶ Processes communicate asynchronously via FIFO queues
- ▶ Each process has exactly one input queue

I/O Elements (1)

SIGNAL A,B;

signal declaration

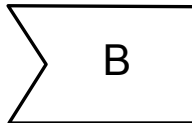
```
SIGNALLIST s1 = A,B;  
SIGNALLIST s2 = s1,C;
```



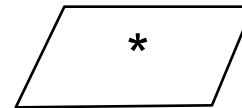
output signal



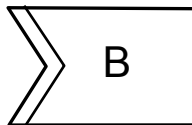
save signal (msg.
remains in queue, no
outgoing transition)



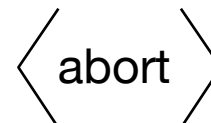
input signal



save all other
signals



priority input



continuous signal with
enabling condition

I/O Elements (2)

Sending to a specific receiver:



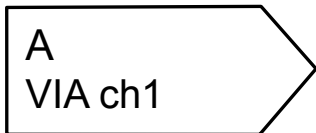
Dest: Process ID



sending a
self-message



sending back
to the sender



sending via a channel

Further addresses:

PARENT

the creating
instance

OFFSPRING

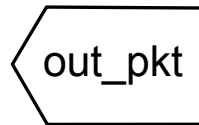
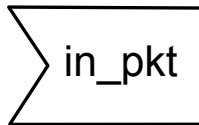
last created instance
by this instance

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



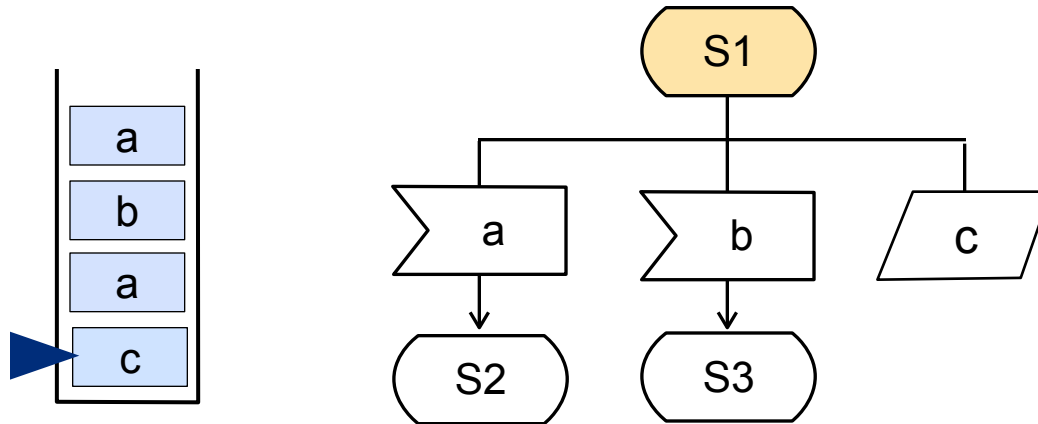
signals from or to processes
logically above or parallel to
this process

pointer or wedge to the right



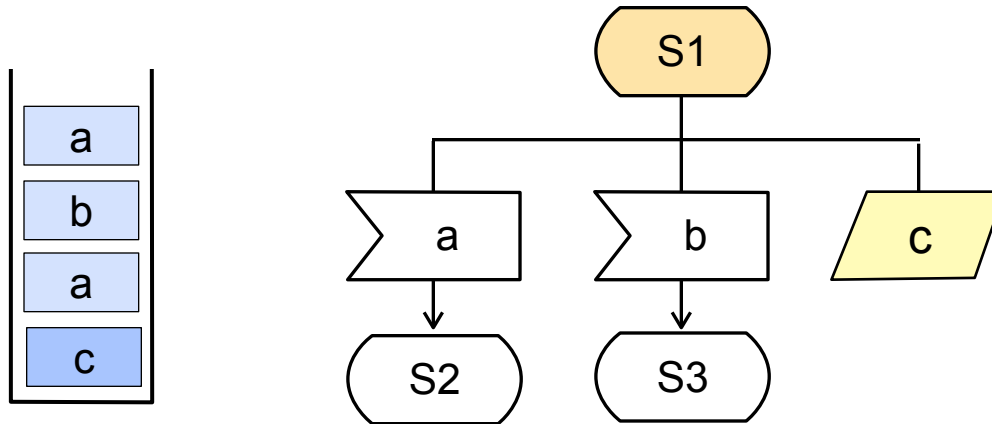
signals from or to processes
logically below this process

Handling signals, Example (1)



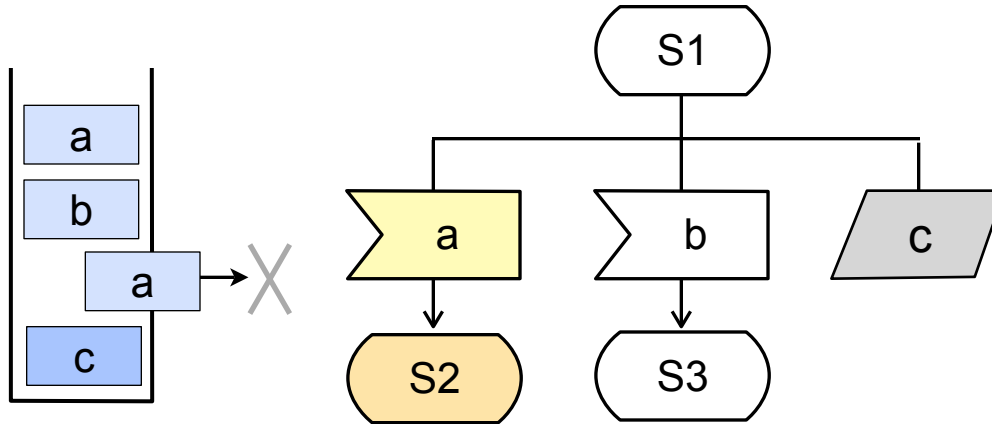
- ▶ The process is in state “S1”
- ▶ Message “c” is first in queue

Handling signals, Example (2)



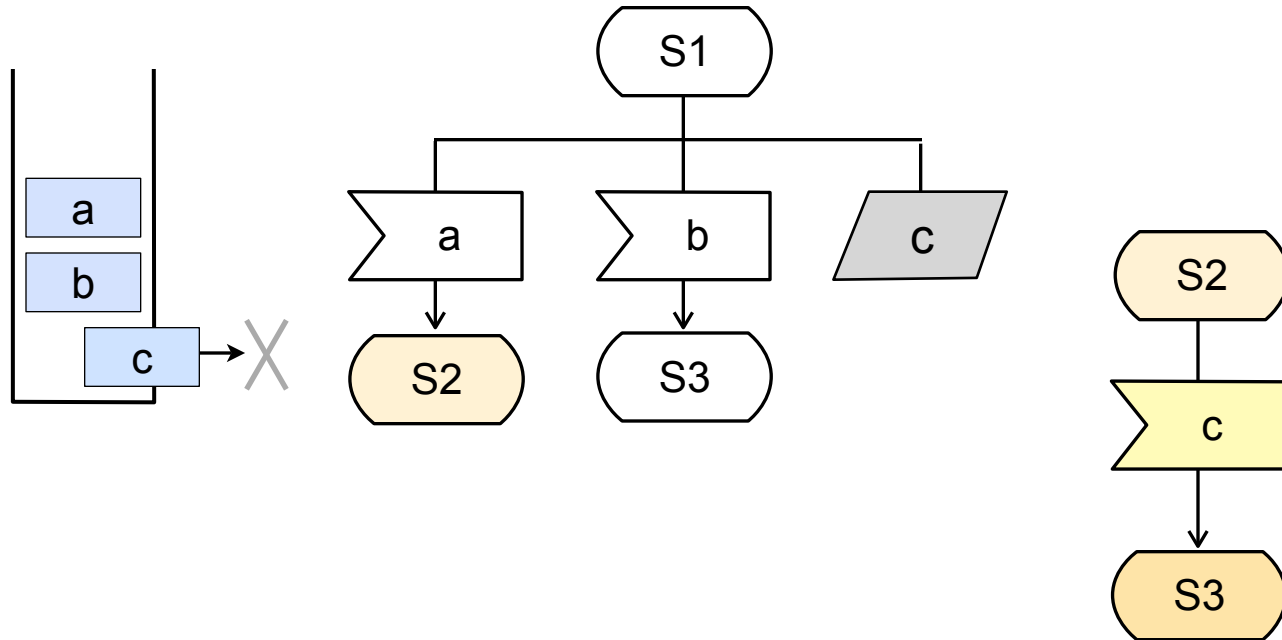
- ▶ “c” is saved and remains ‘passively’ in queue

Handling signals, Example (3)



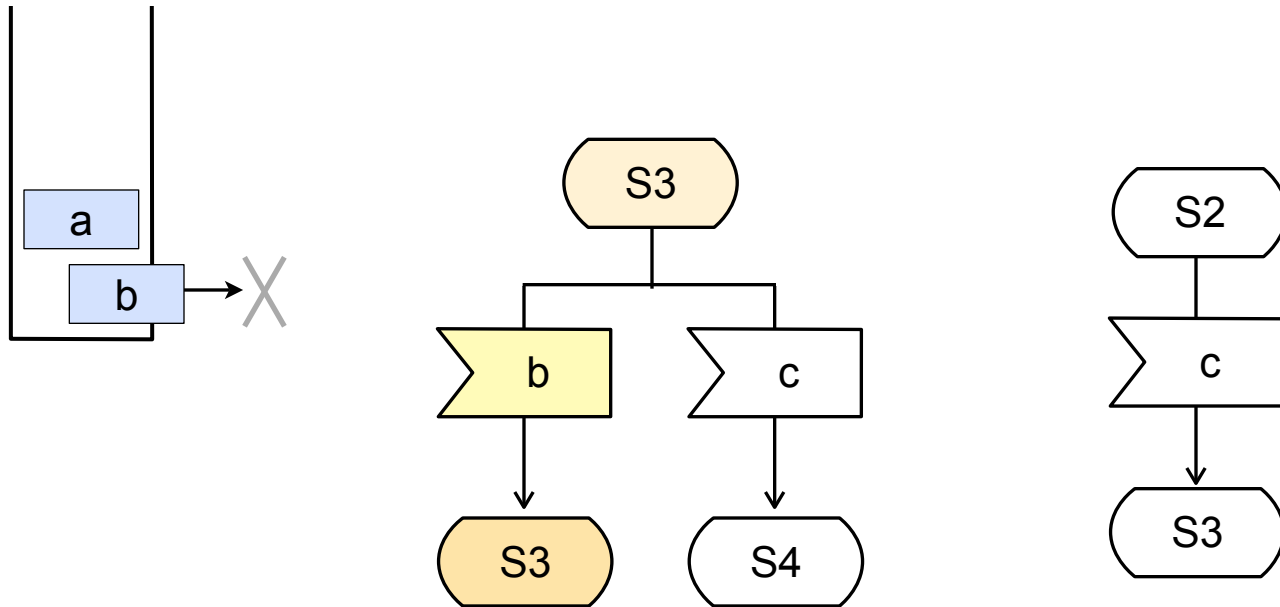
- ▶ “a” is consumed and removed from the queue
- ▶ It triggers the transition to S2

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:



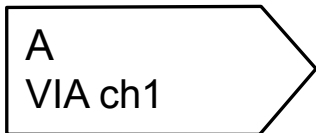
Dest: Process ID



sending a
self-message



sending back
to the sender



sending via a channel

Further addresses:

PARENT

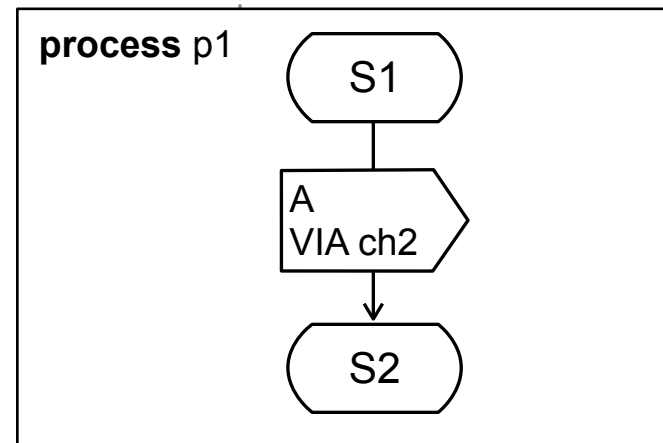
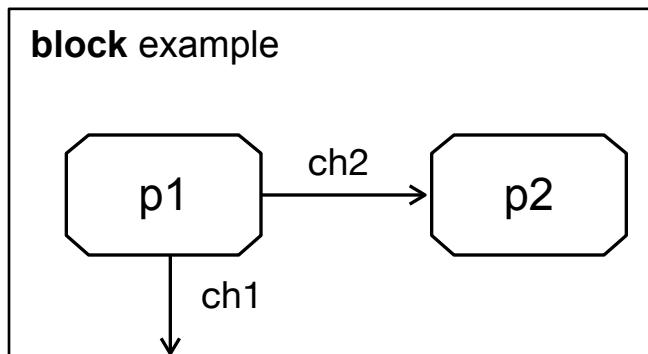
the creating
instance

OFFSPRING

last created instance
by this instance

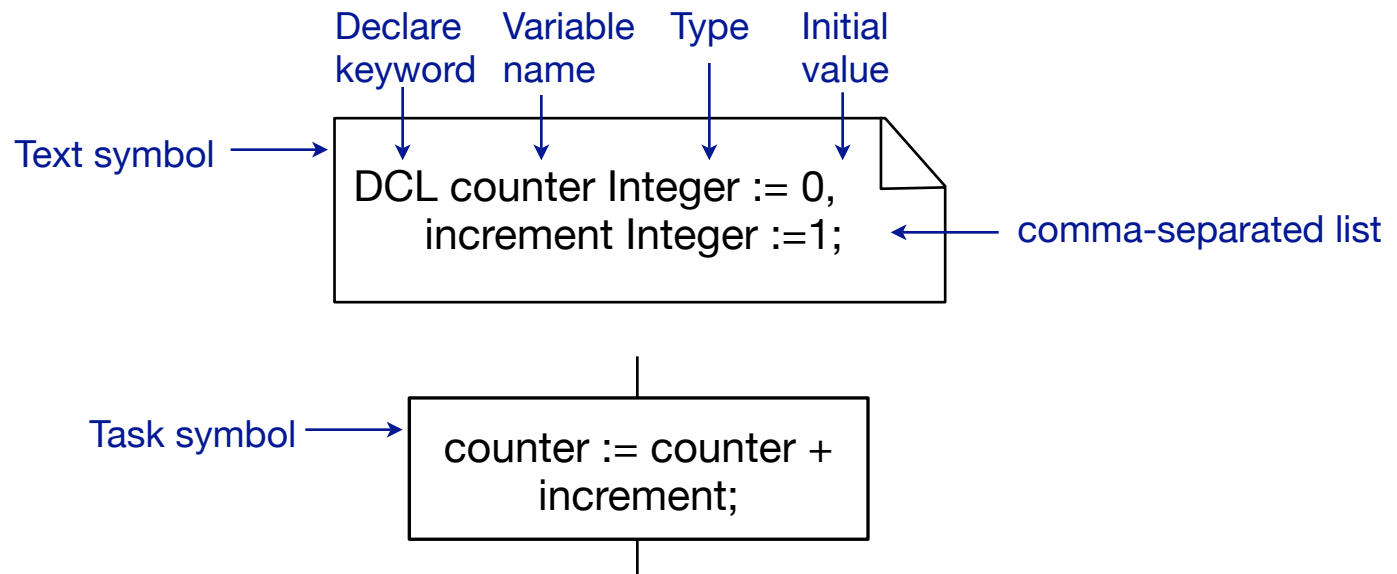
Sending signals, Example

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



Variables

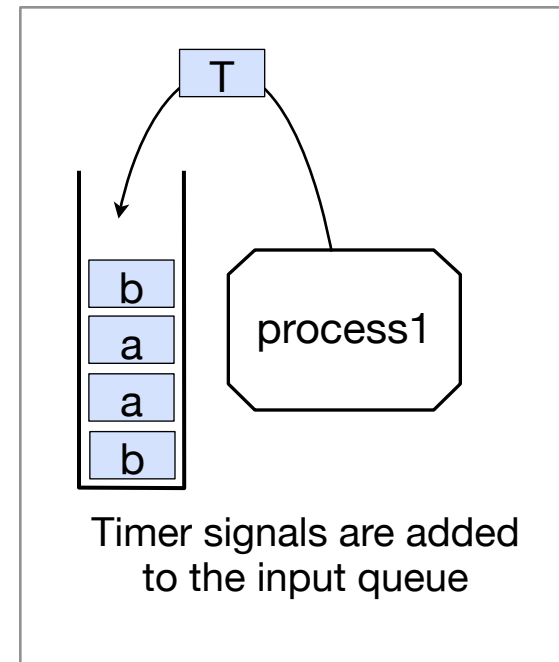
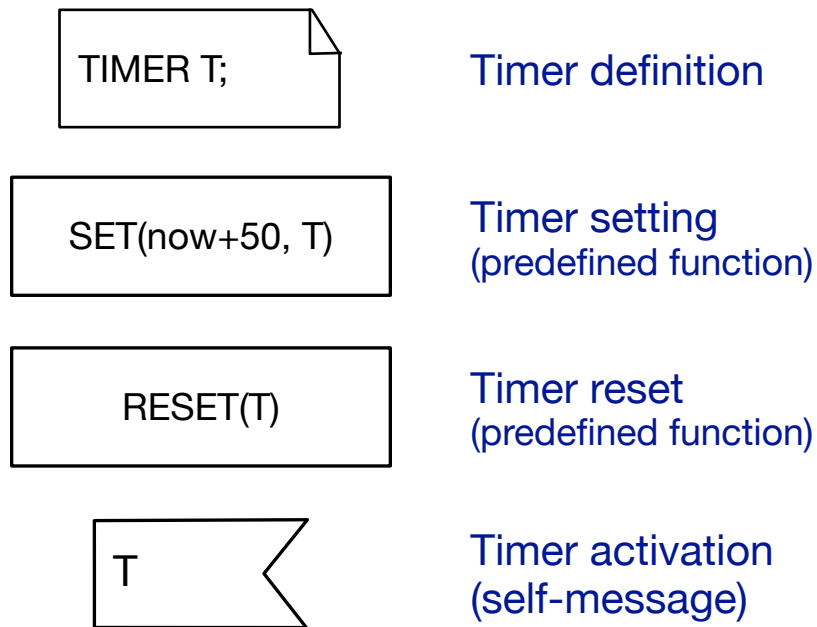
- ▶ Variables are declared in a text symbol
- ▶ They are manipulated in an task



[sdl-forum.org/sdl88tutorial/]

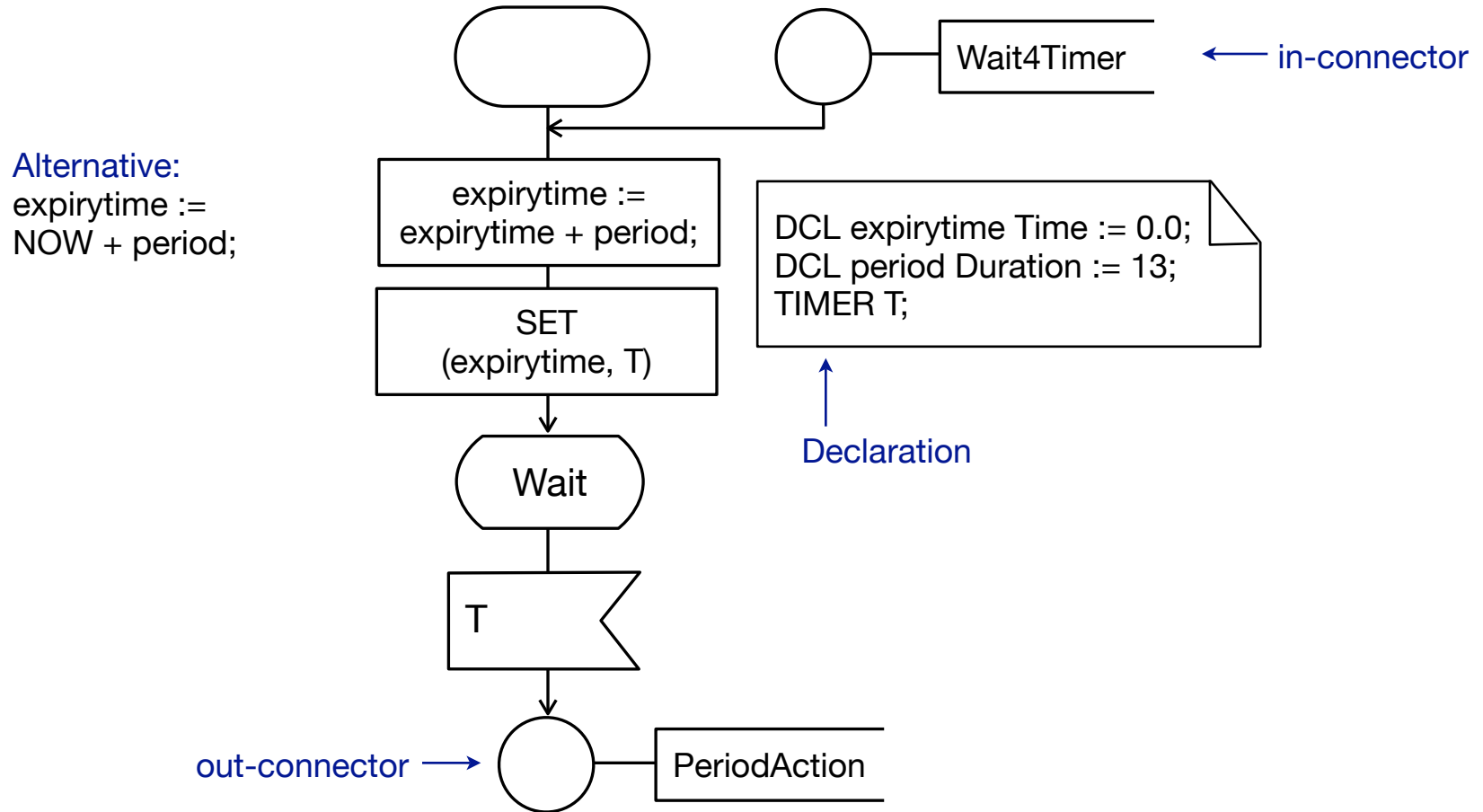
Timers

- ▶ Timers are self-messages which are added to the input queue



[sdl-forum.org/sdl88tutorial/]

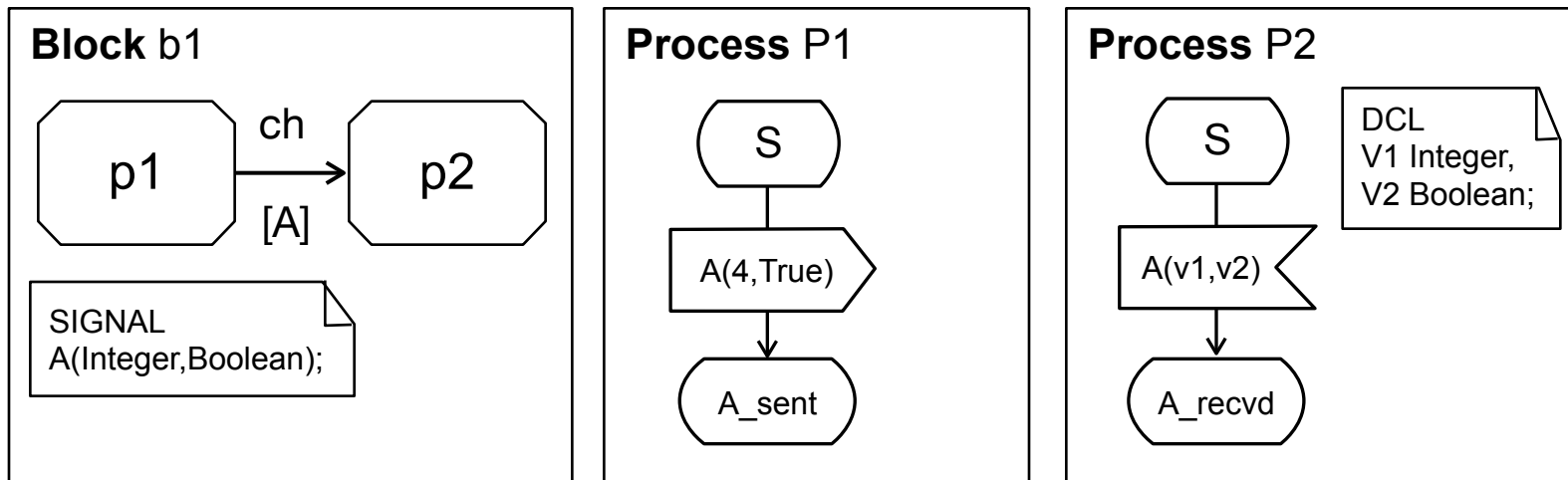
Timers, Example



[sdl-forum.org/sdl88tutorial/]

Passing data variables

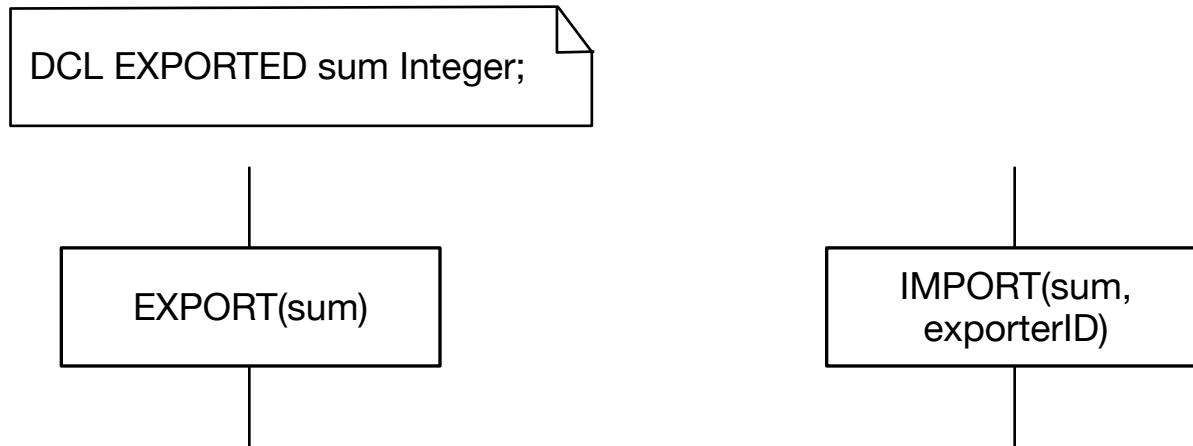
- ▶ Signals can contain data values
- ▶ Input and output must be compatible



[sdl-forum.org/sdl88tutorial/]

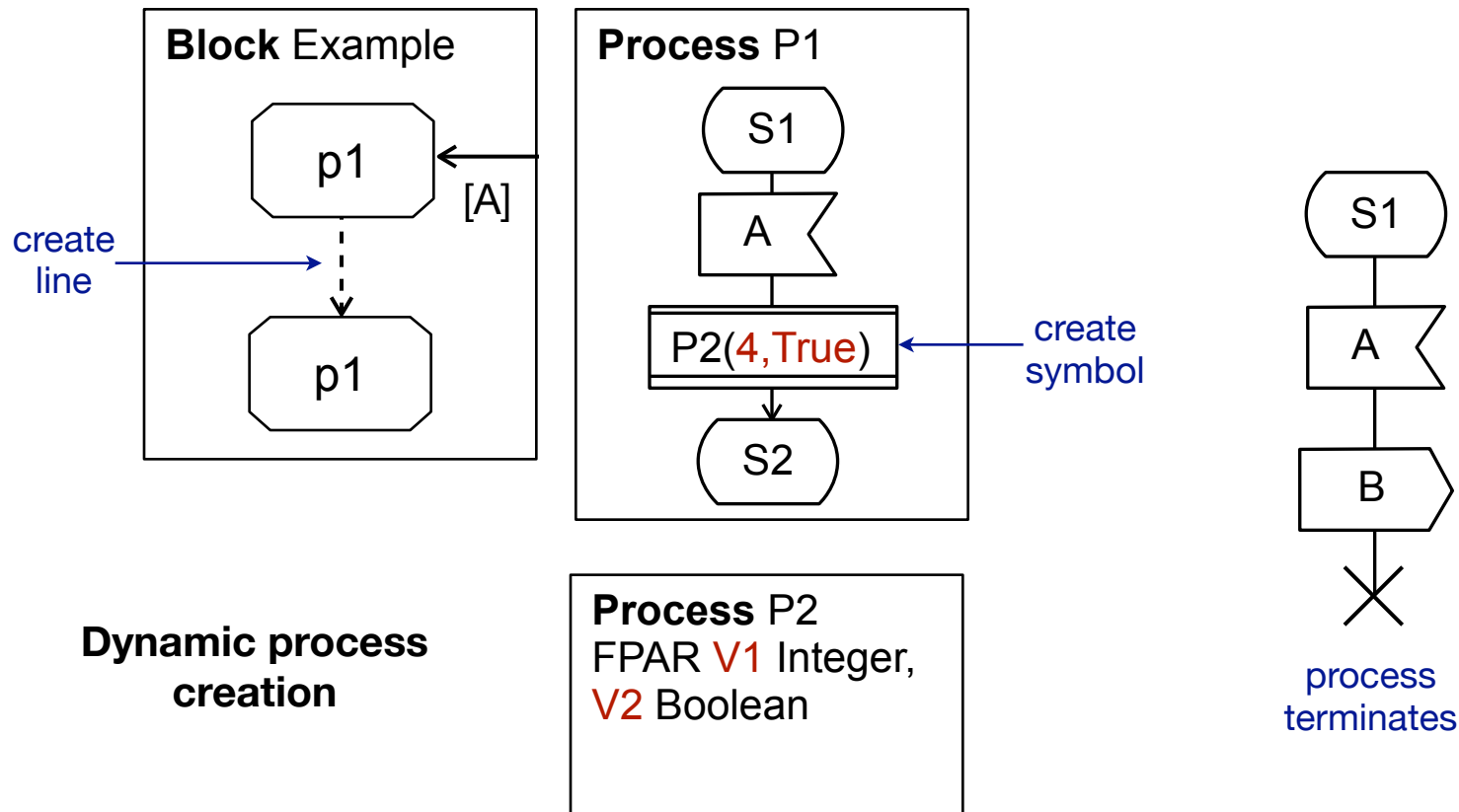
Import and Export of Variables

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



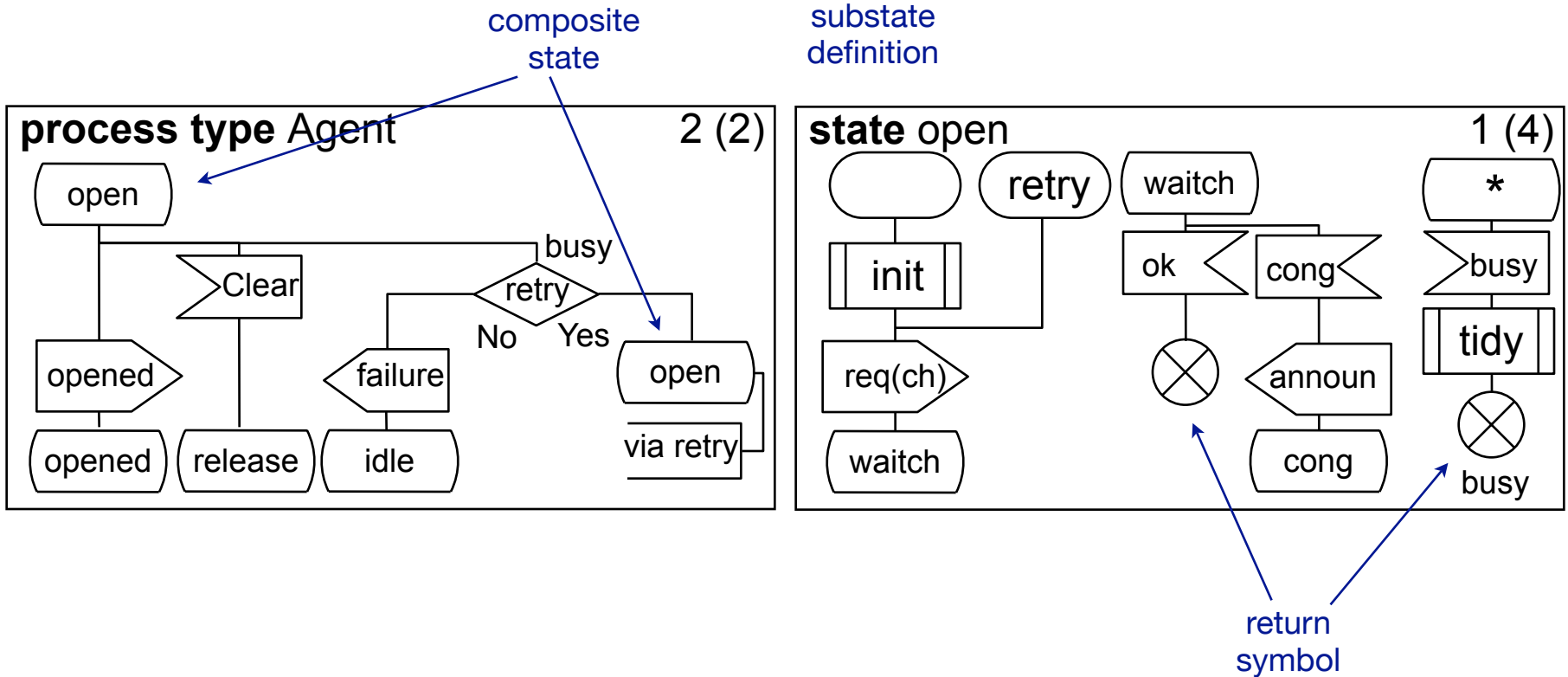
[sdl-forum.org/sdl88tutorial/]

Process creation and termination



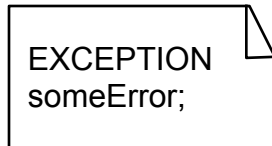
[sdl-forum.org/sdl88tutorial/]

Composite States

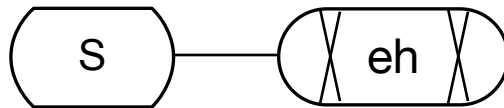


[R. Reed, SDL-2000 Presentation, sdl-forum.org/sdl2000present/]

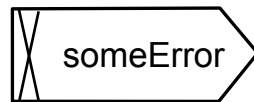
Exceptions



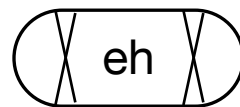
Exception definition



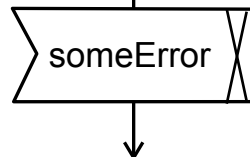
State with associated exception handler



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/]

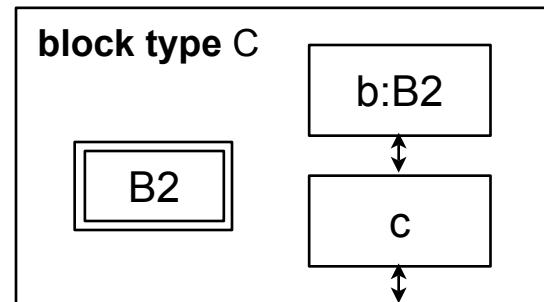
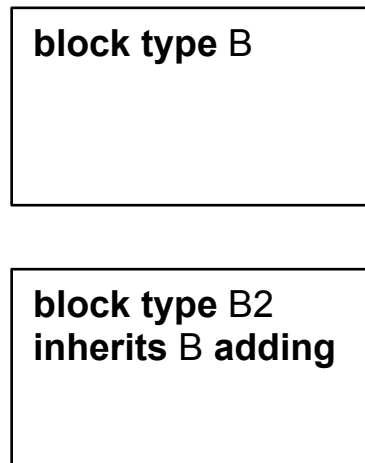
Data Types, Example

```
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 .);
```

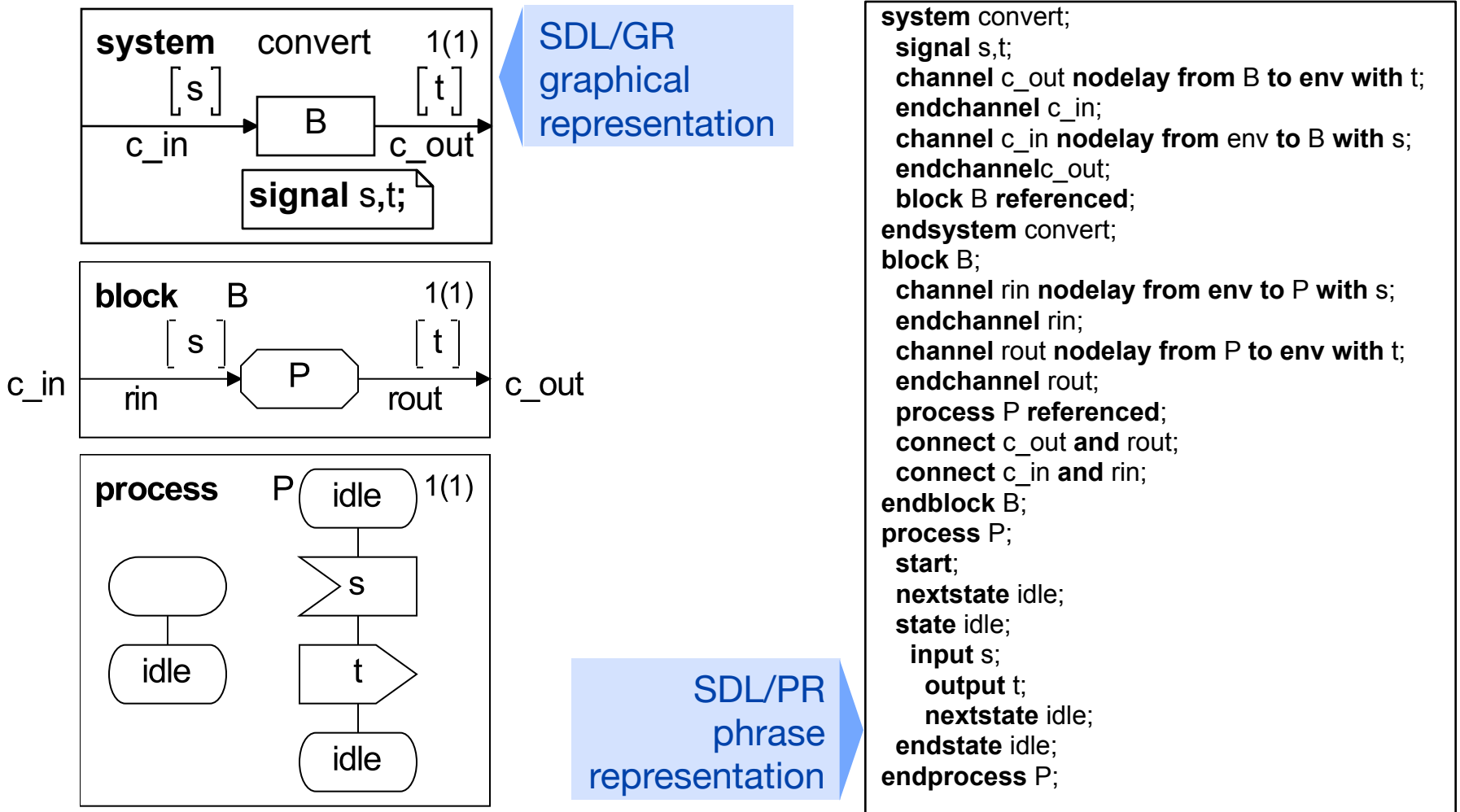
[R. Reed, SDL-2000 Presentation, sdl-forum.org/sdl2000present/]

Object orientation

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



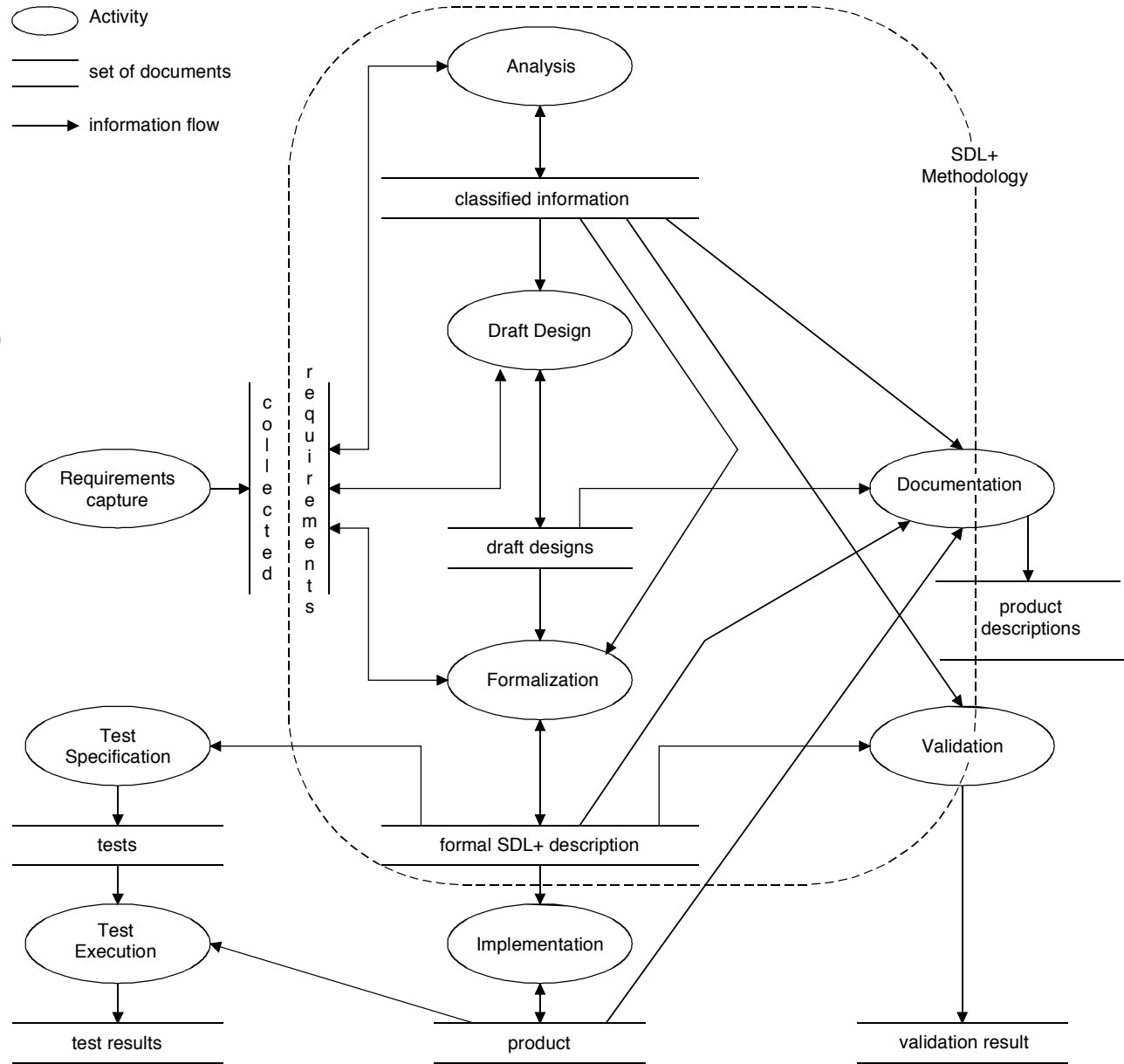
SDL/GR vs. SDL/PR



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

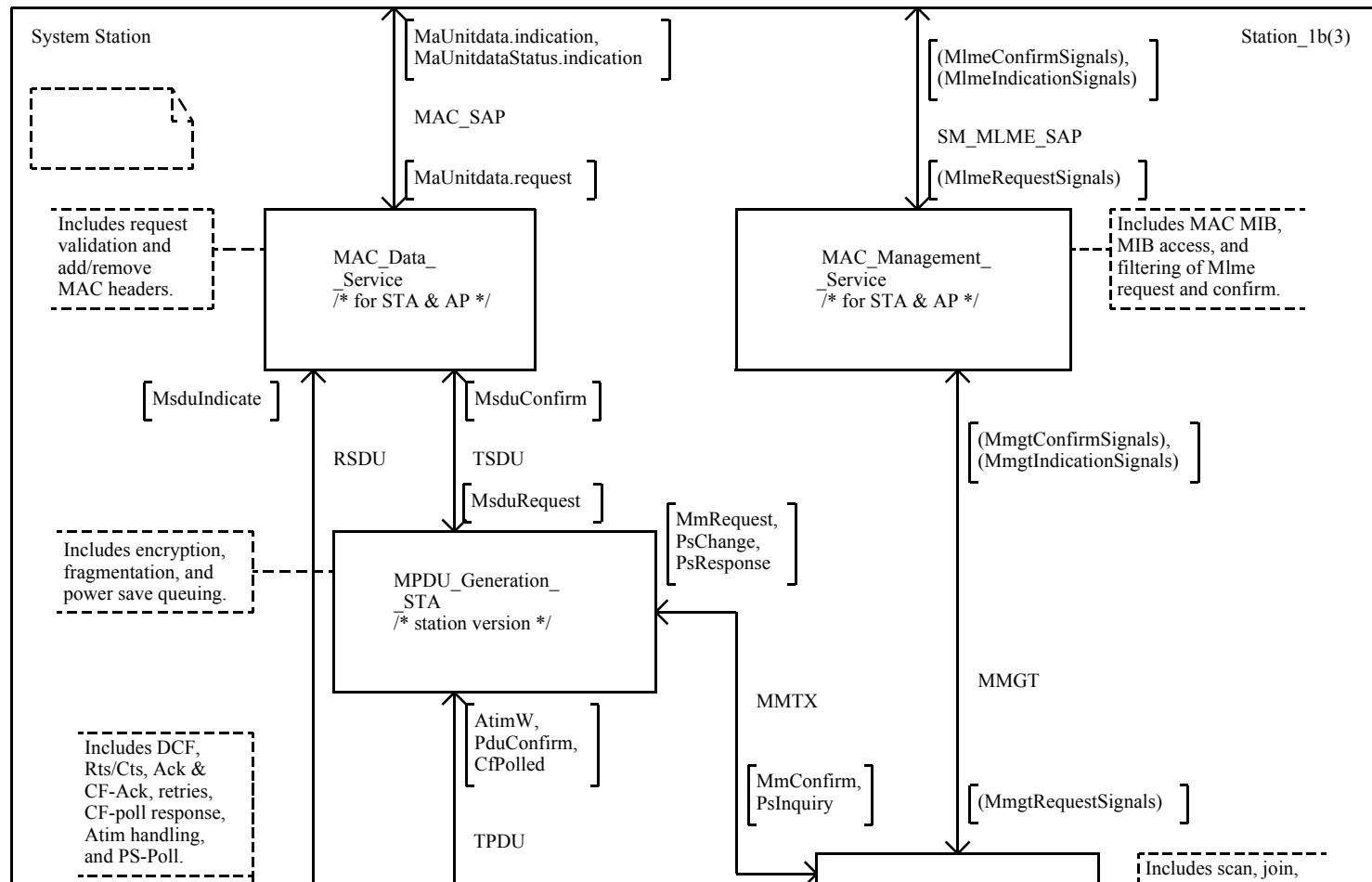
SDL in the development process

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



[ITU-T Z.100 Supplement 1]

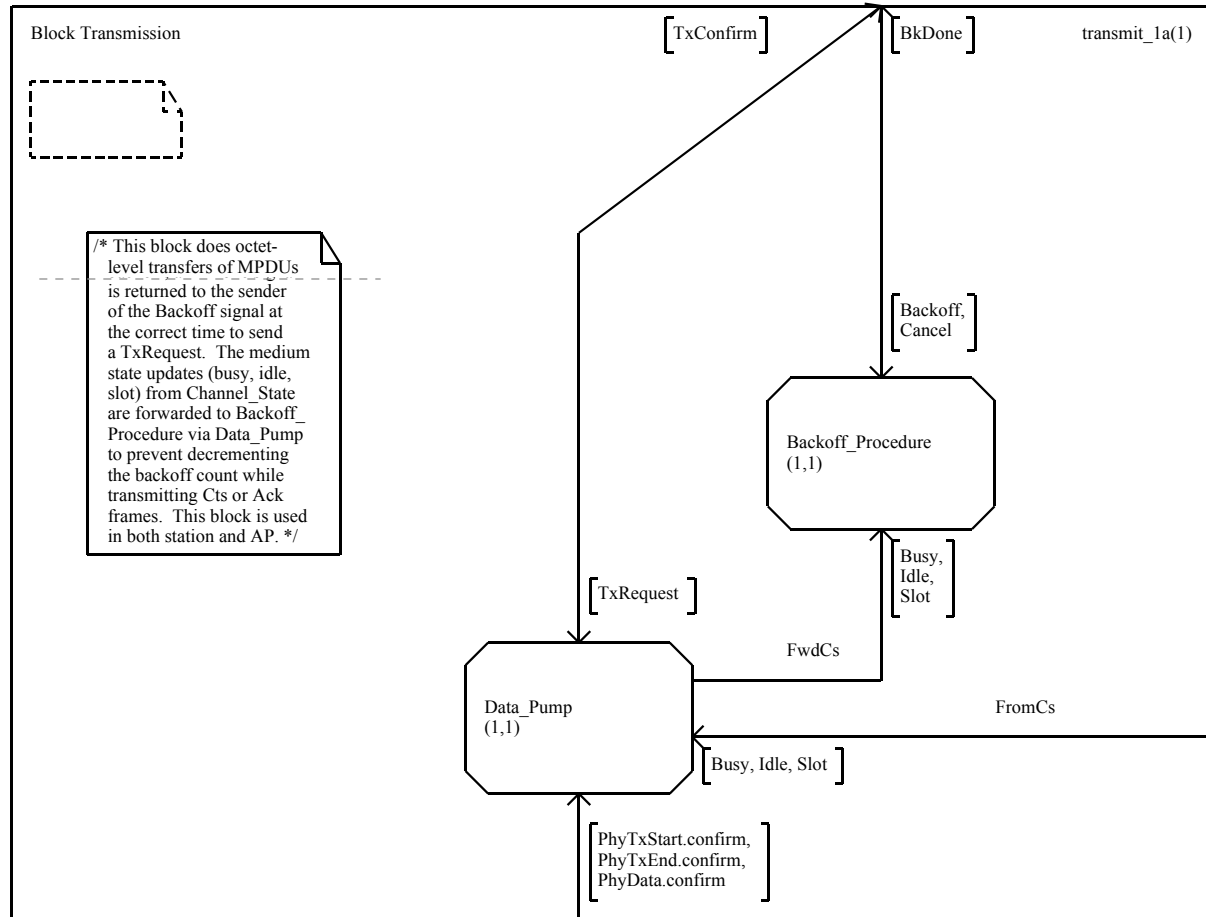
SDL in practice: 802.11 Specification



System specification (part)

[IEEE Std. 802.11-2007]

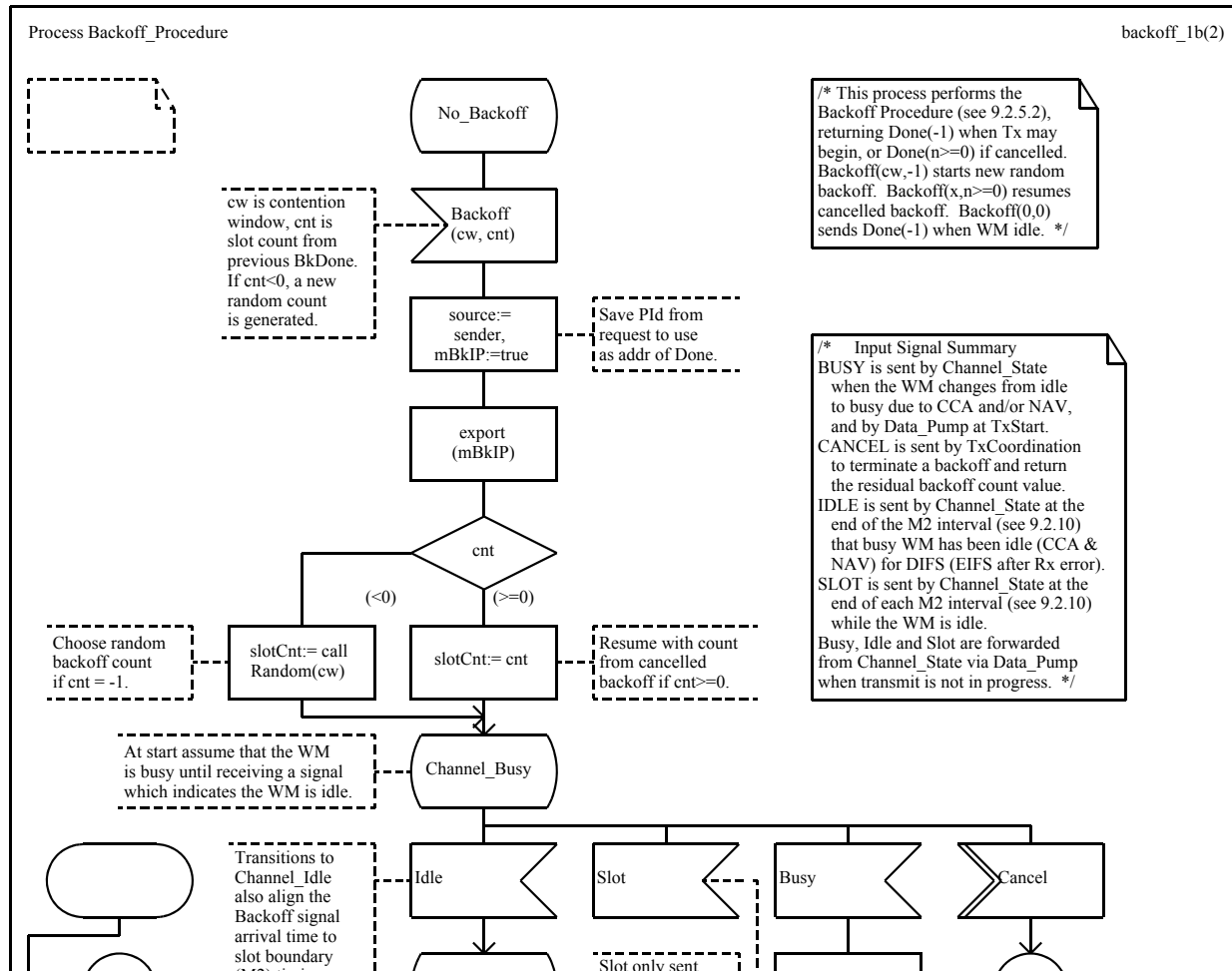
SDL in practice: 802.11 Specification



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

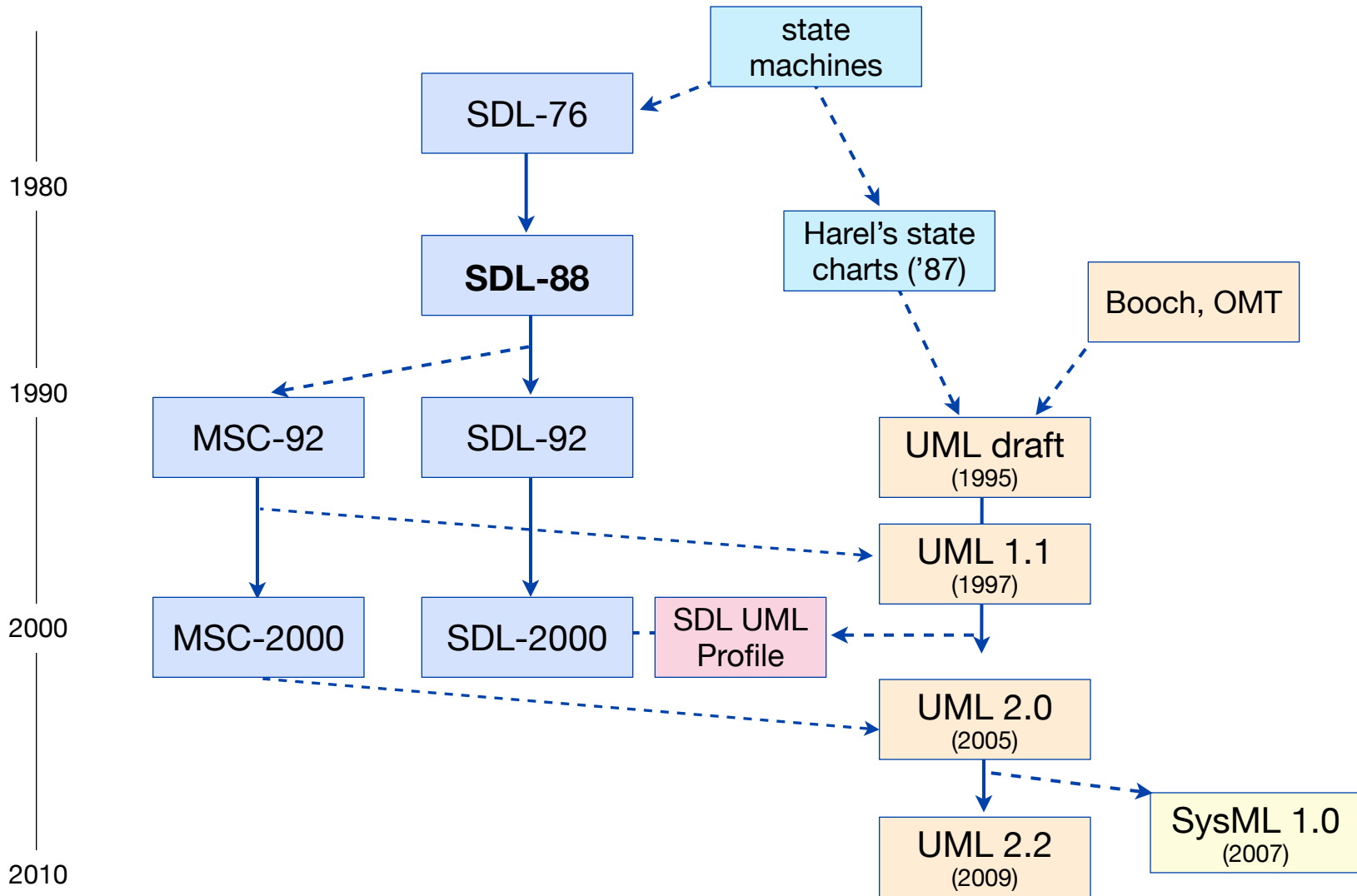
[R. Reed, "Notes on SDL-2000 for the new millennium, Computer Networks (35), 2001]

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

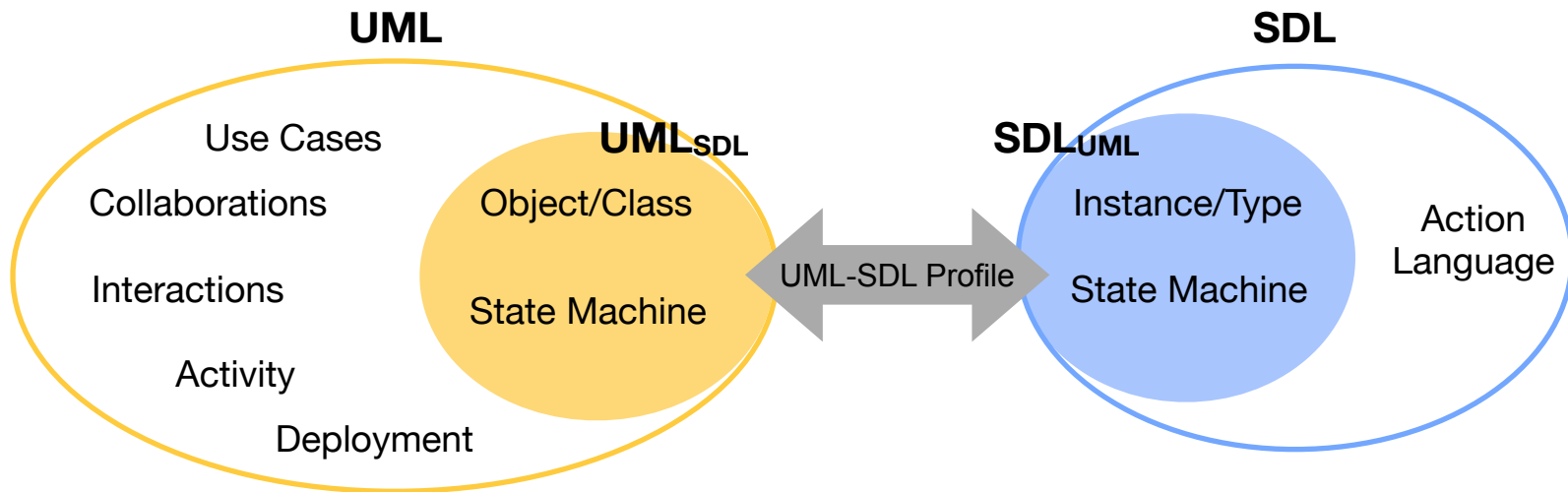


SDL and UML

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

B. Møller-Pedersen: "SDL combined with UML", Teletronikk 4.2000]

SDL and UML



Mapping subsets of UML and SDL

[B. Møller-Pedersen: "SDL combined with UML", Teletronikk 4.2000]

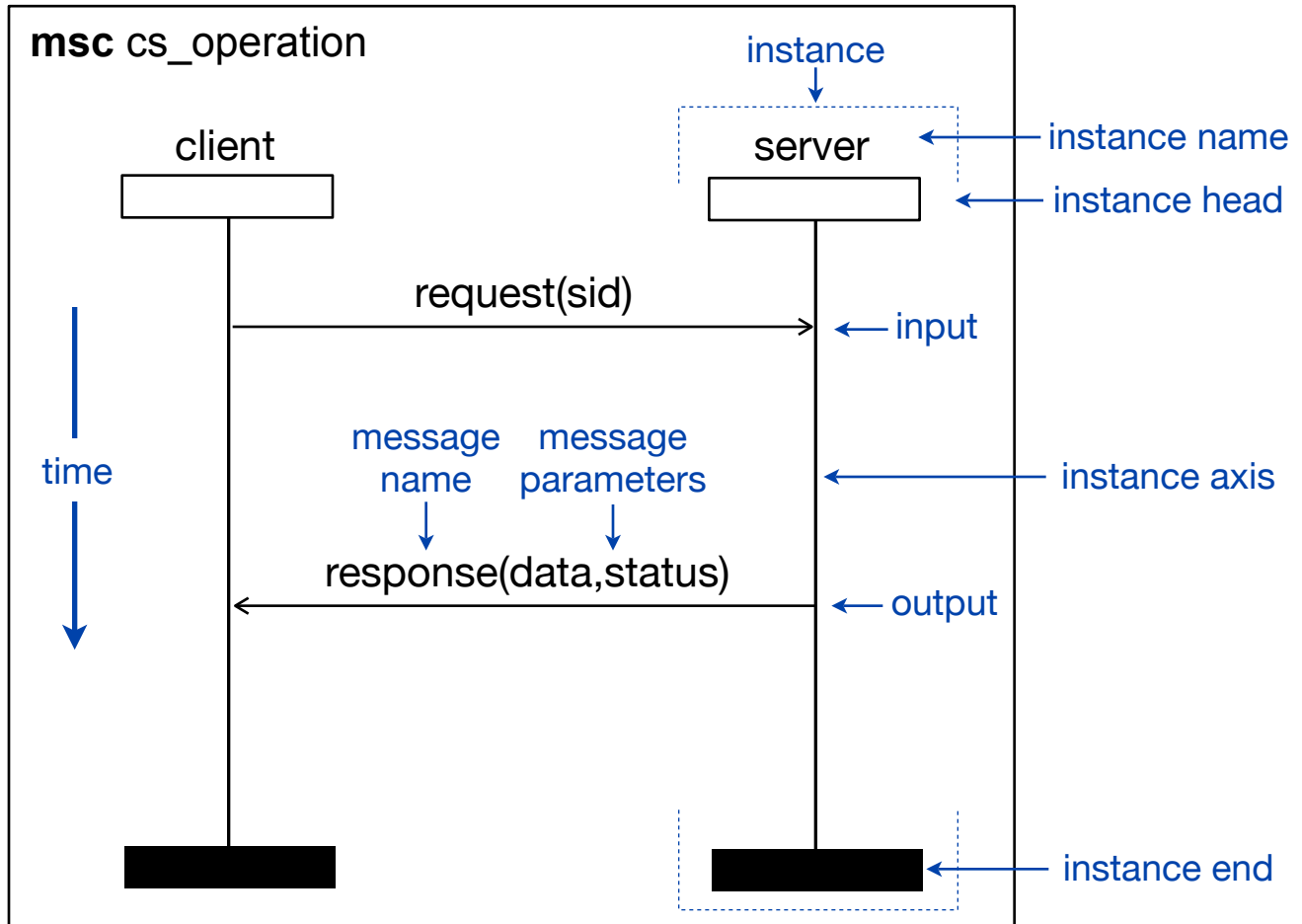
Lessons learned

- ▶ 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

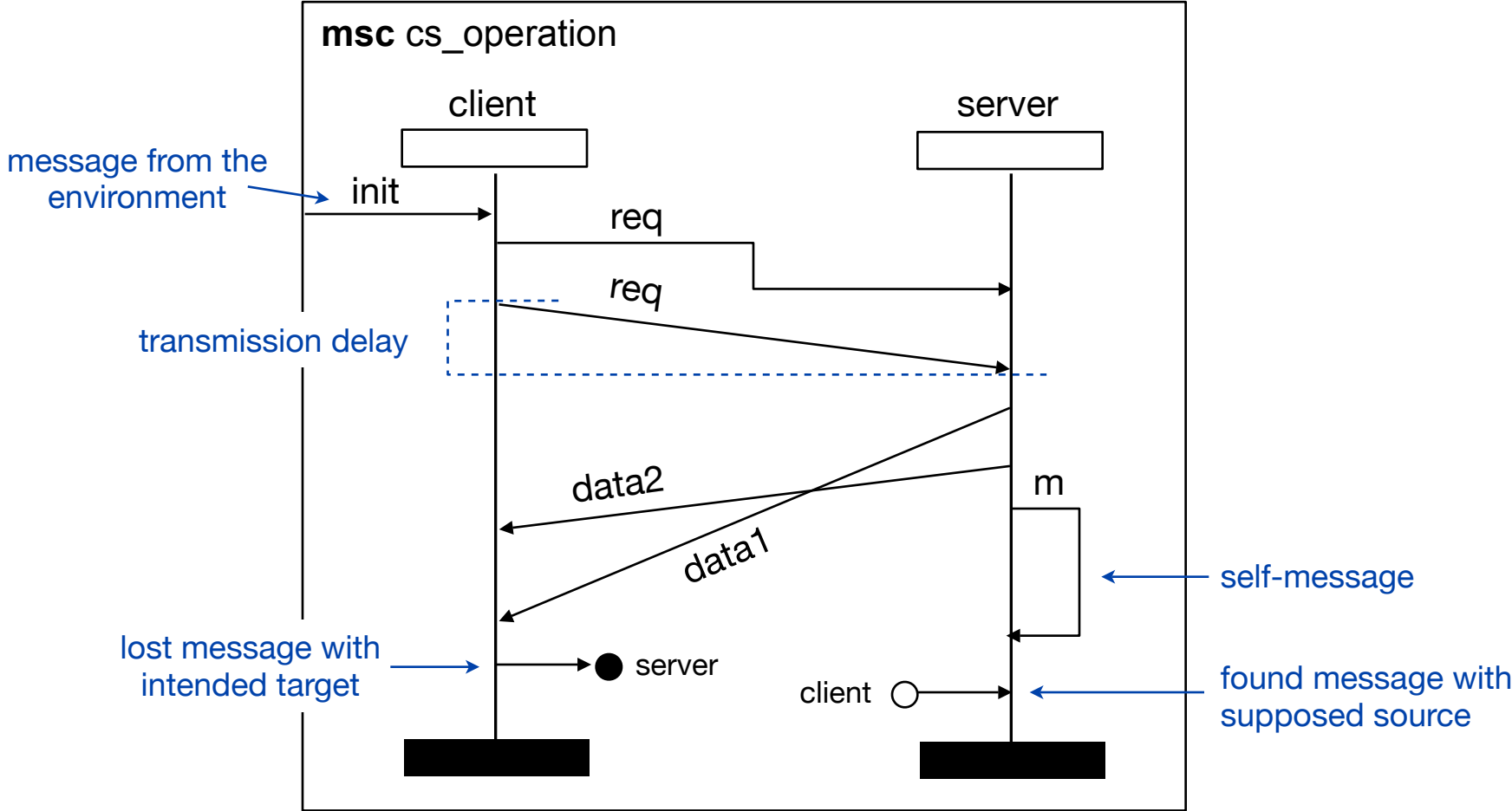
Message Sequence Charts

- ▶ 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



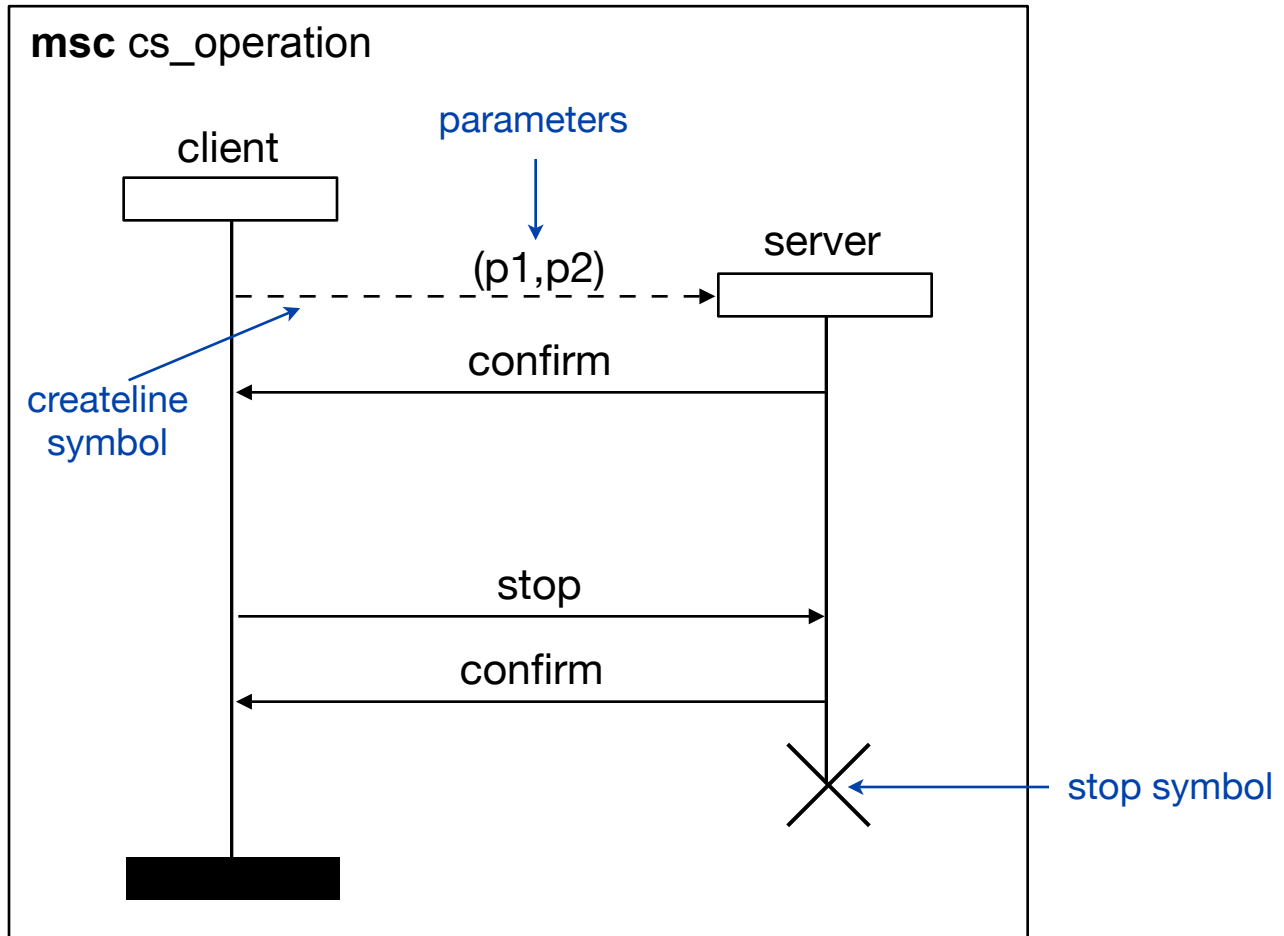
MSC Basics: Messages



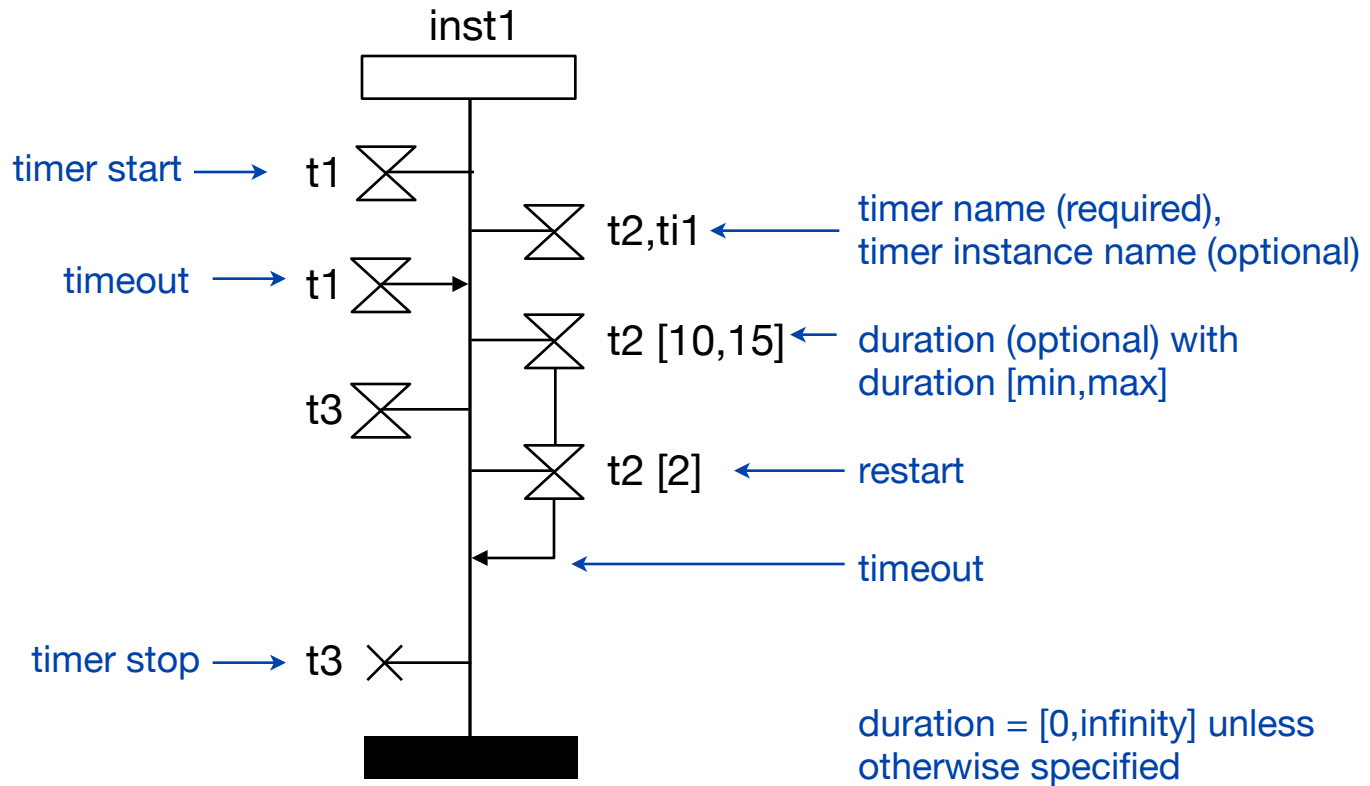
MSC Basic assumptions

- ▶ 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)

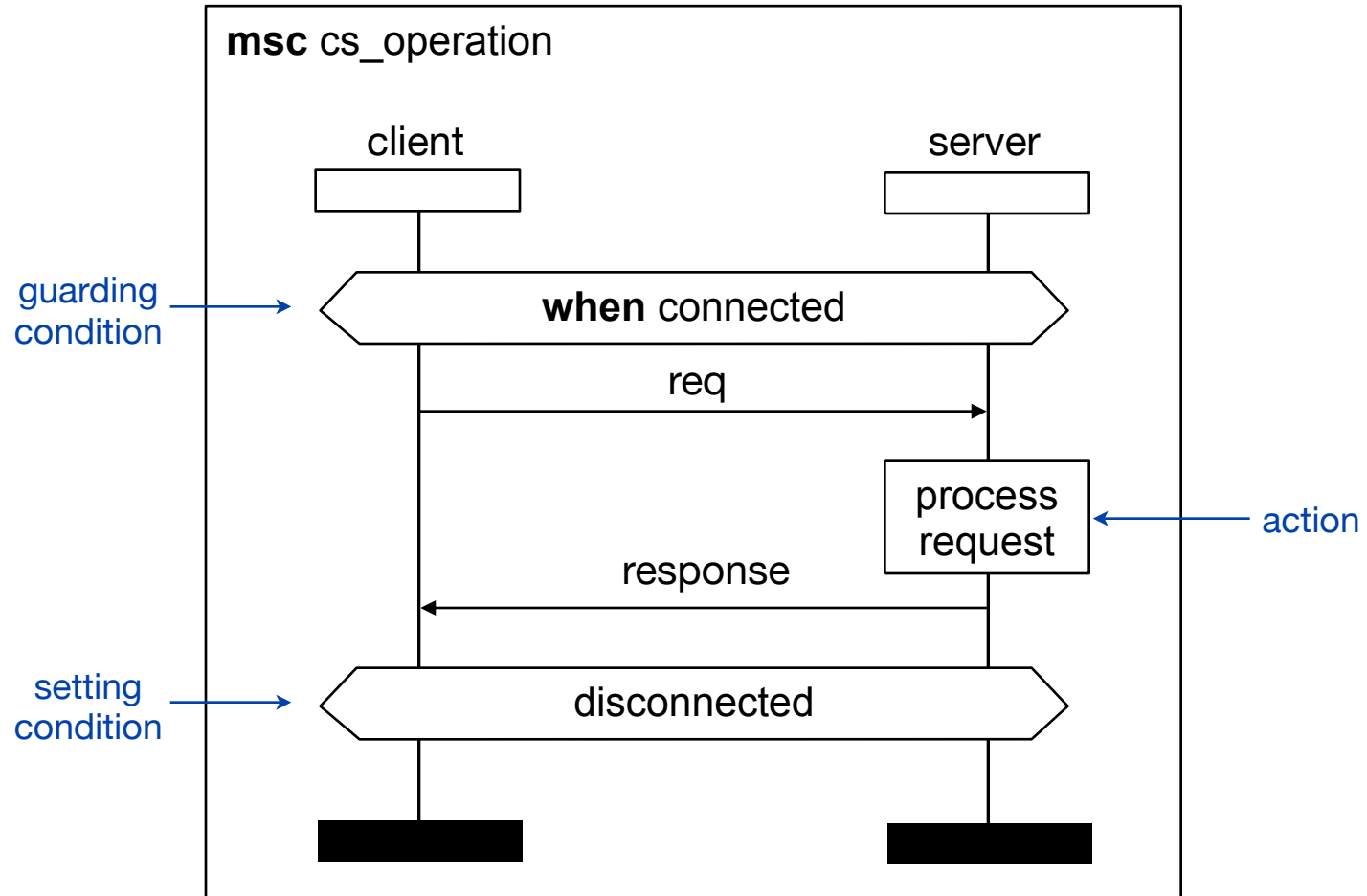
Instance creation and termination



Timers



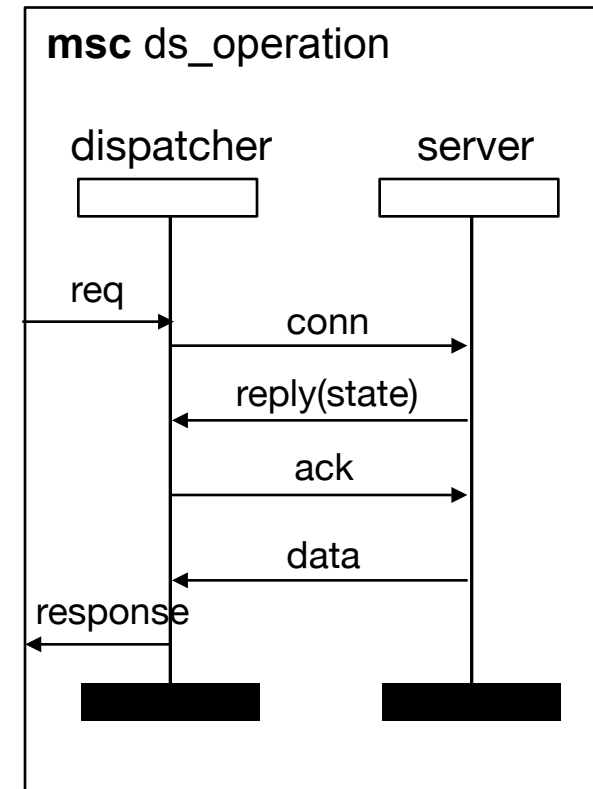
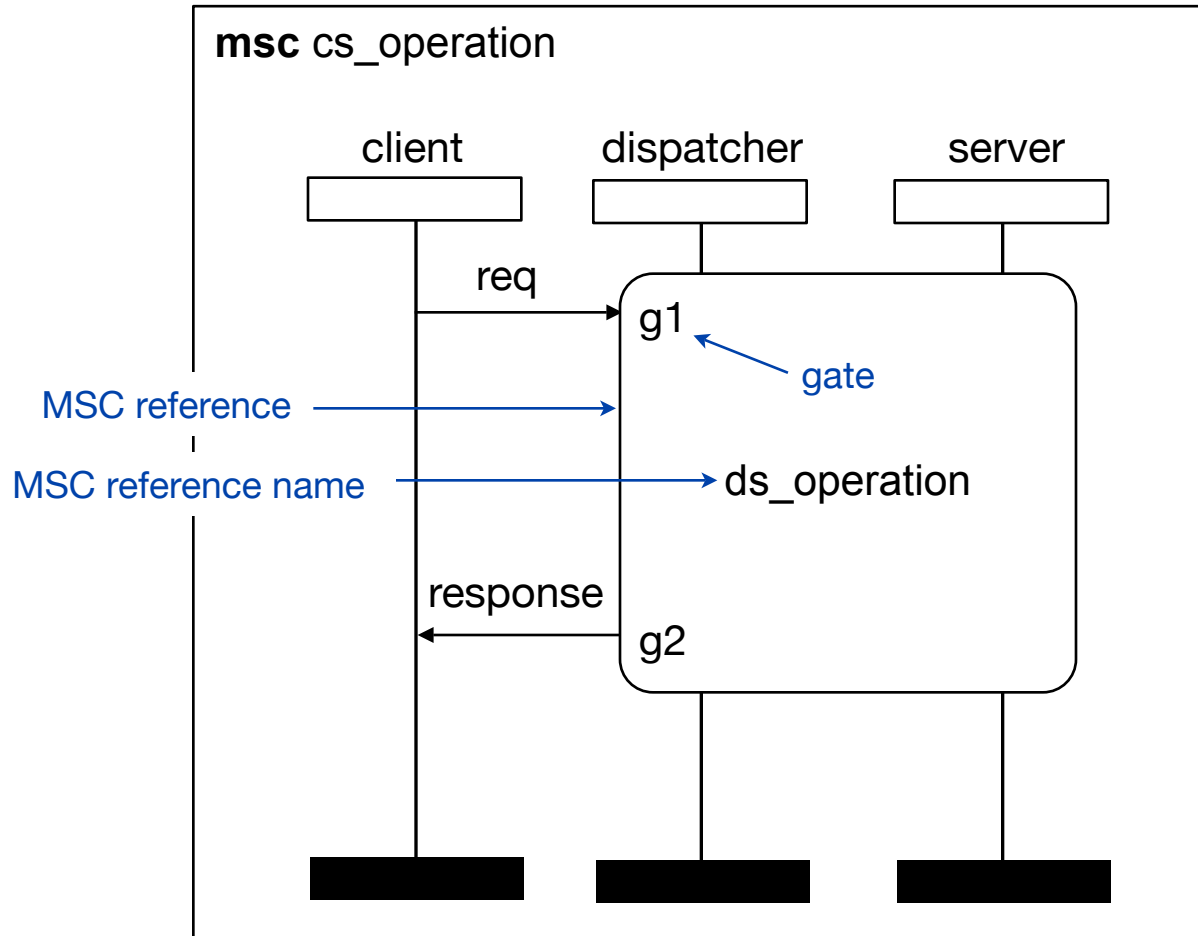
Conditions



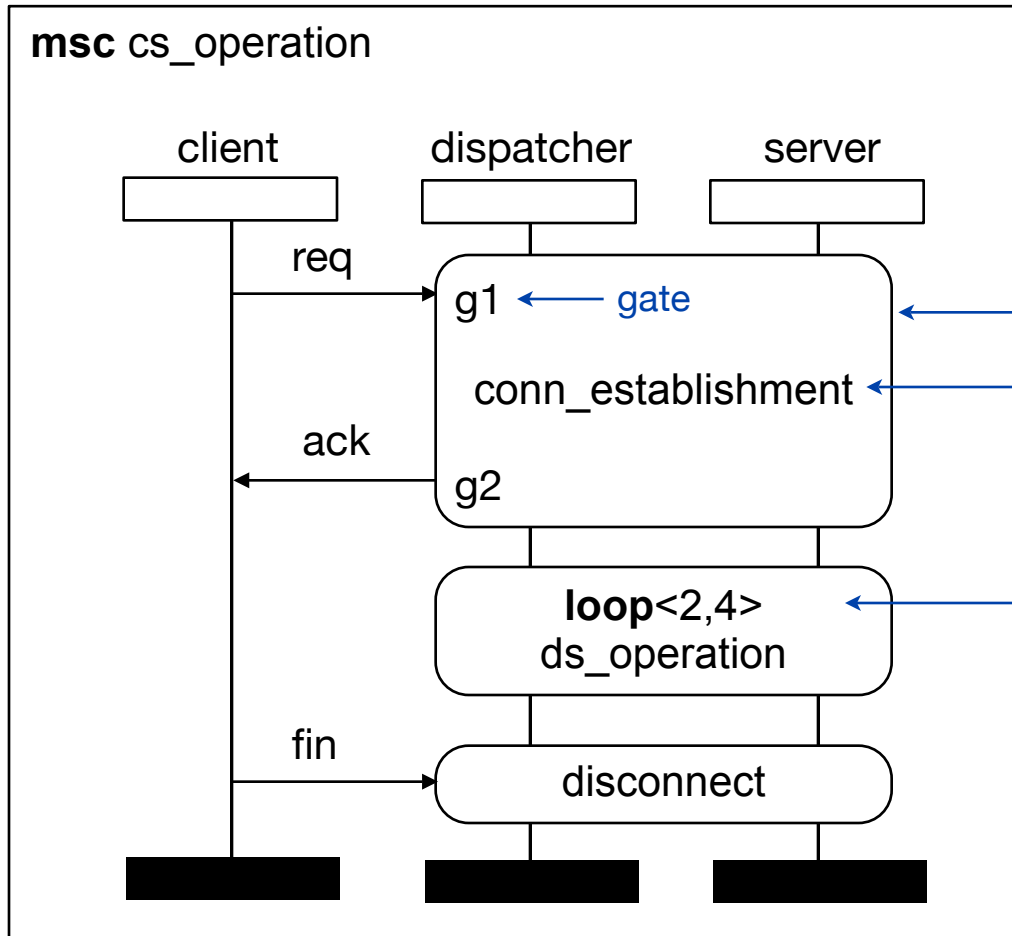
Conditions

- ▶ 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)

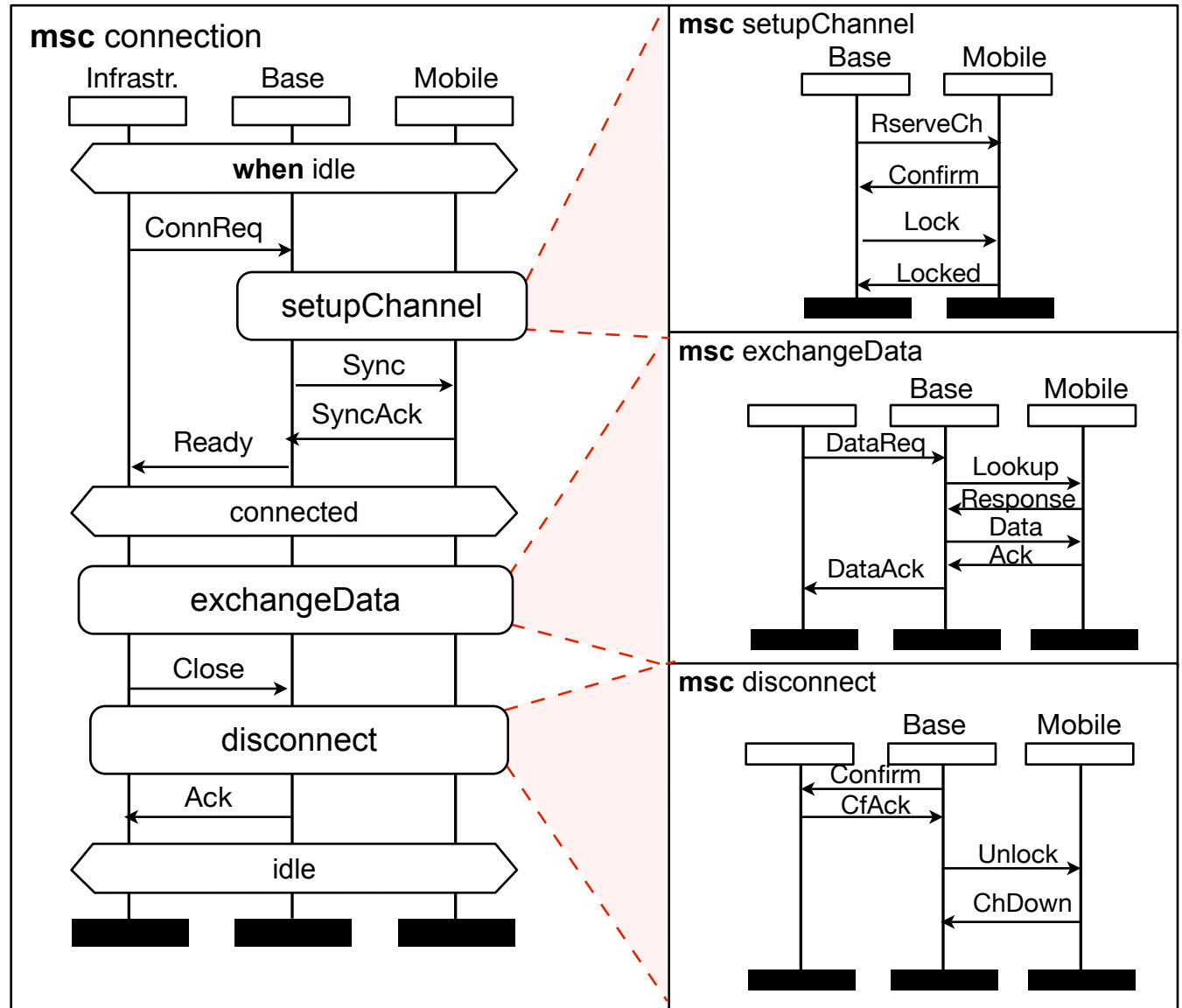
MSC Reference



MSC Reference

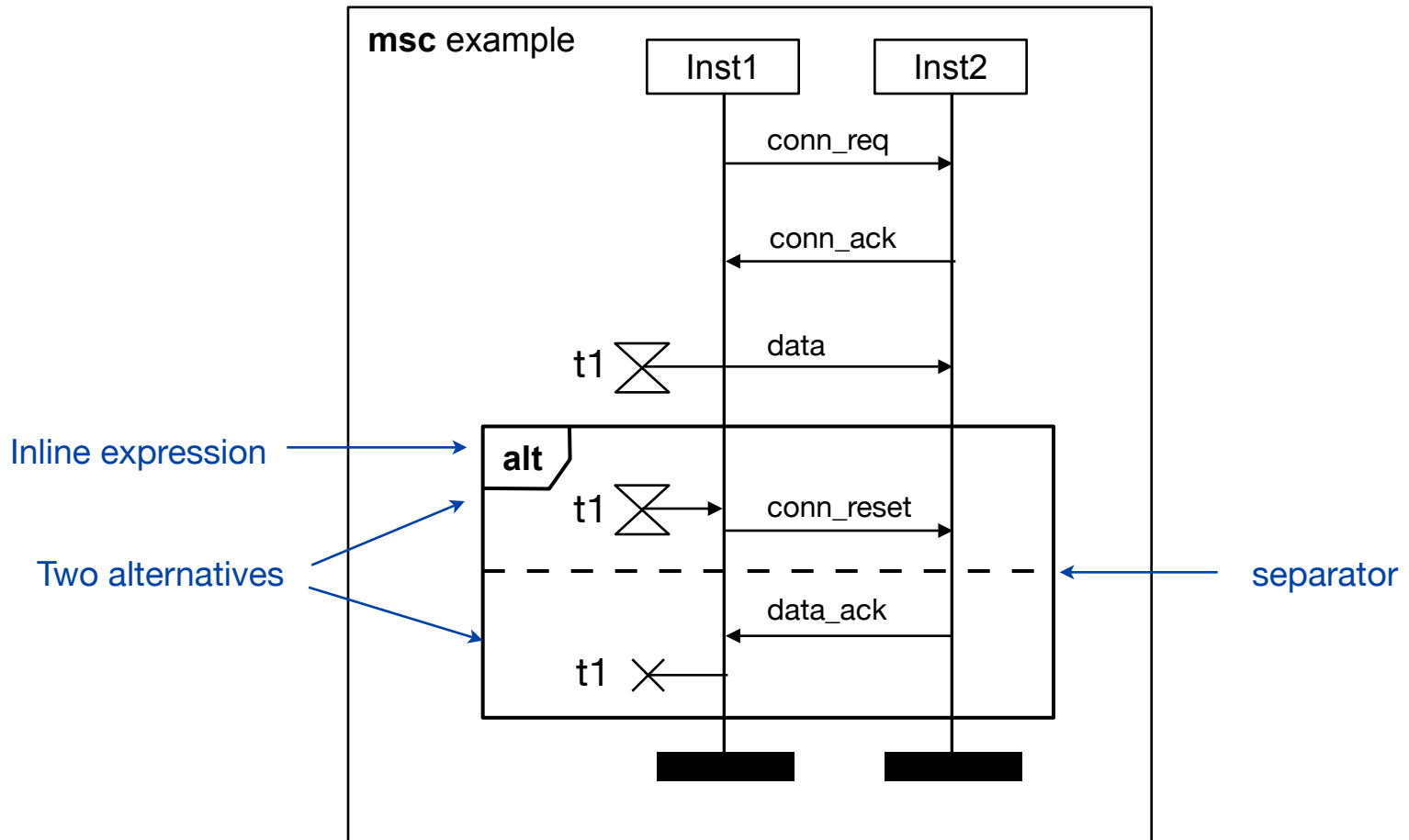


MSC Reference Example

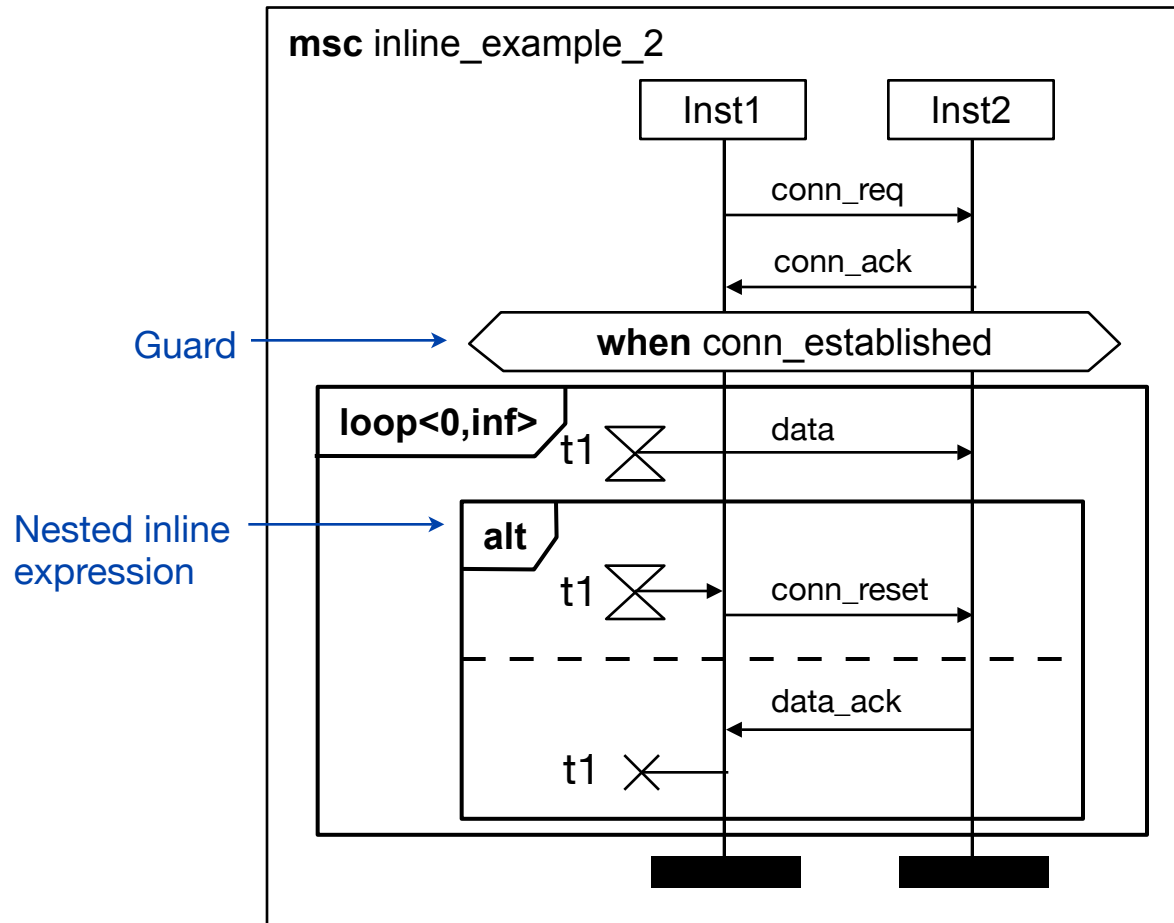


[Grabowski, Reed: "ASN.1, MSC, SDL and TTCN Today", Tutorial, WITUL 2004]

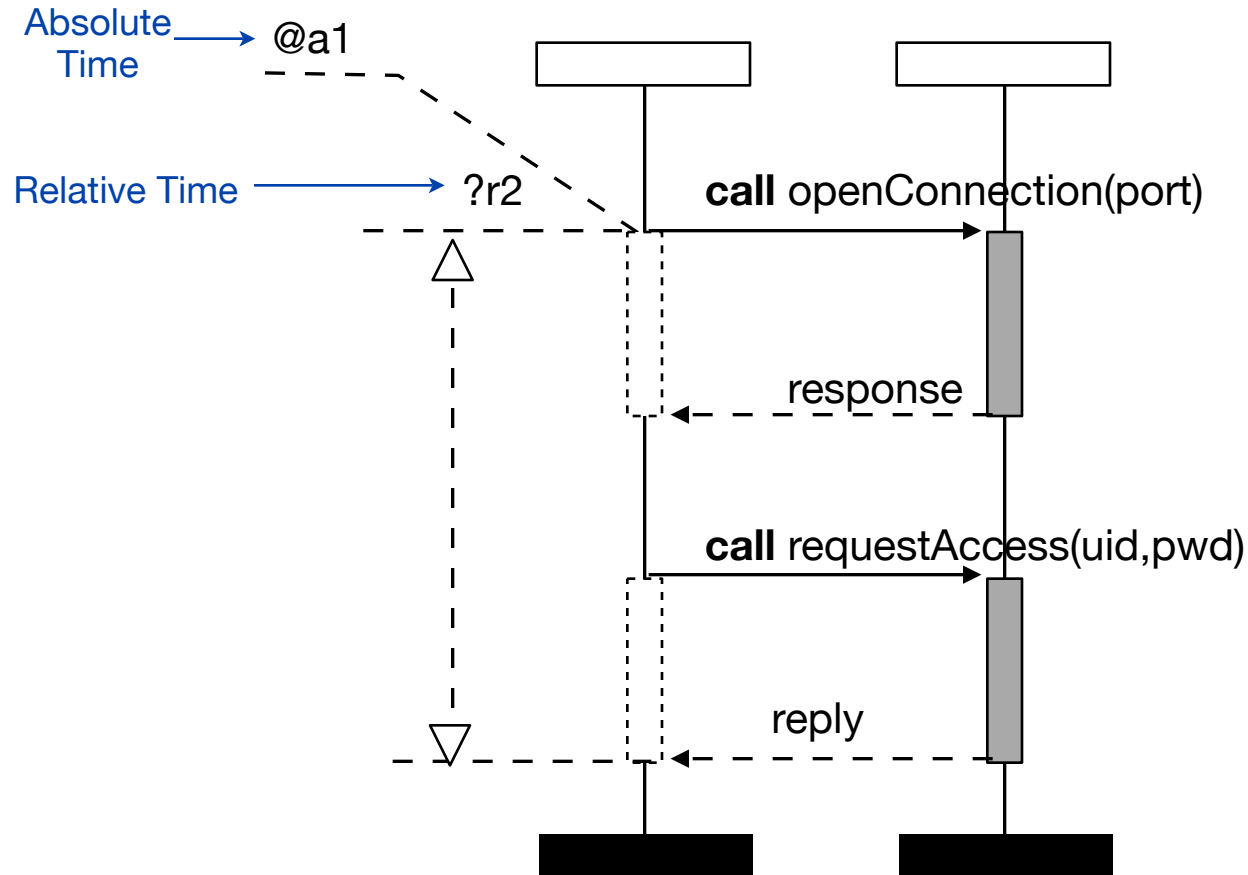
Inline expressions



Nested and guarded inline expressions

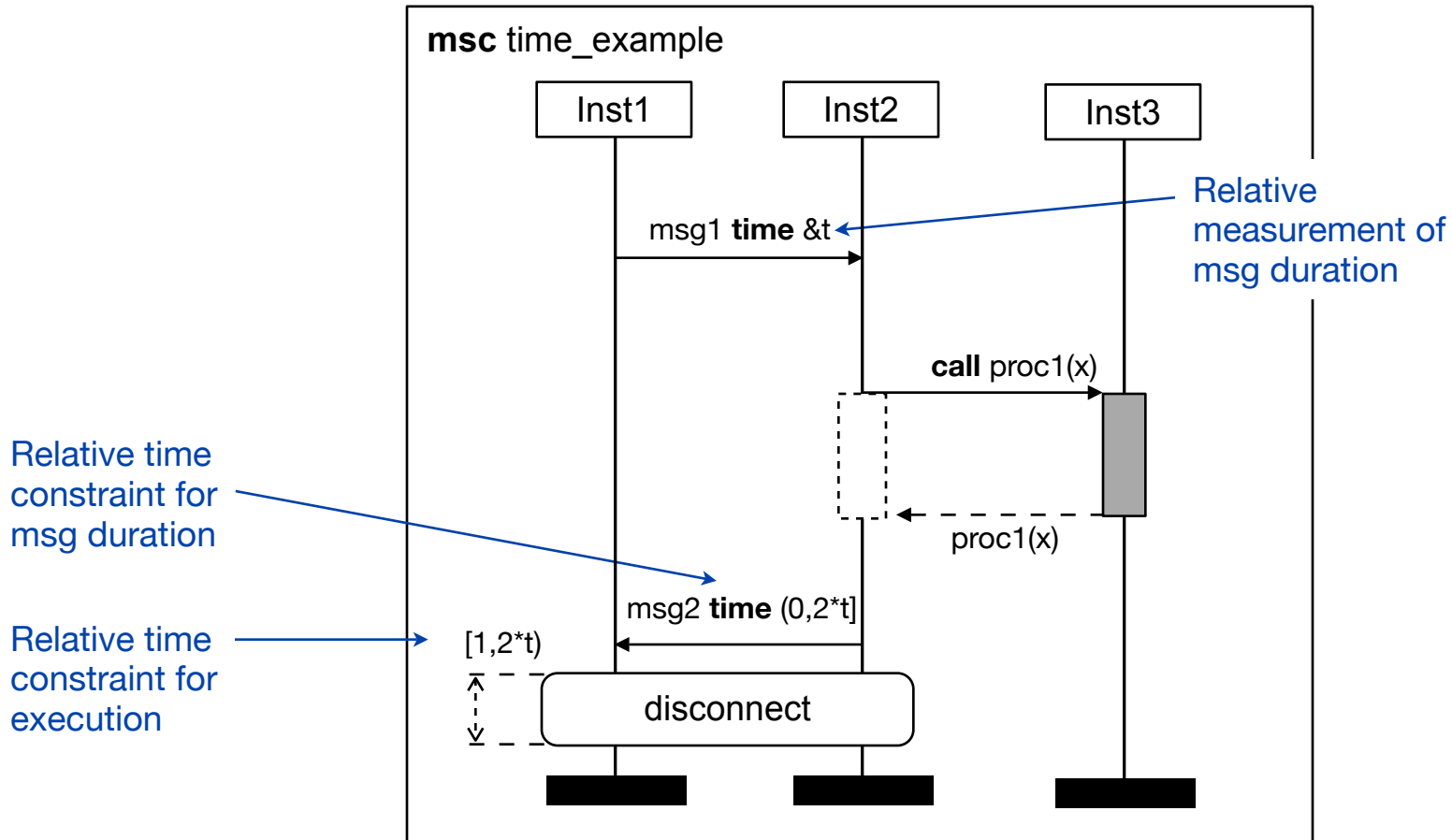


Time observation



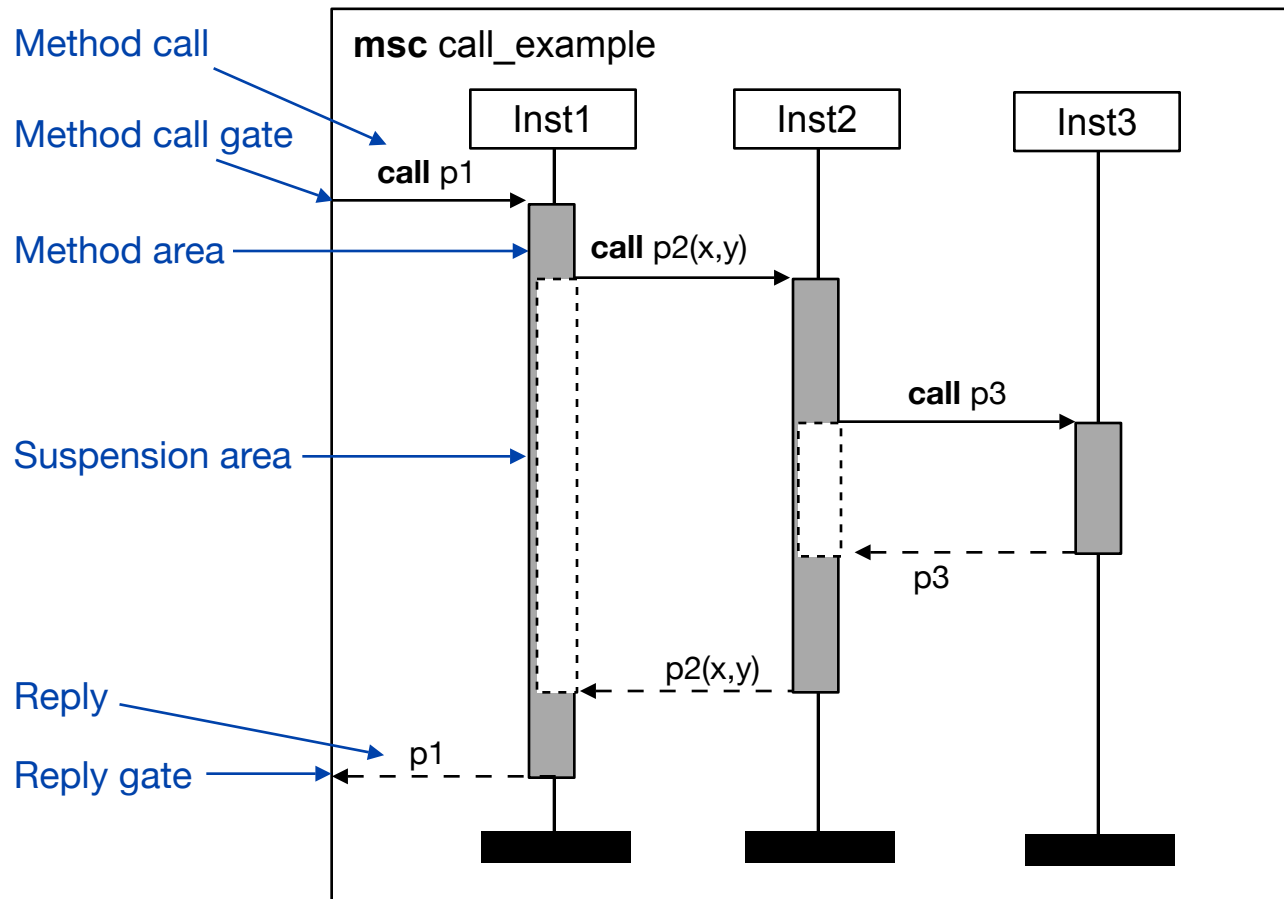
[Ø. Haugen: "MSC-2000 Interaction for the new Millenium", sdl-forum.org/MS2000present]

Time constraints

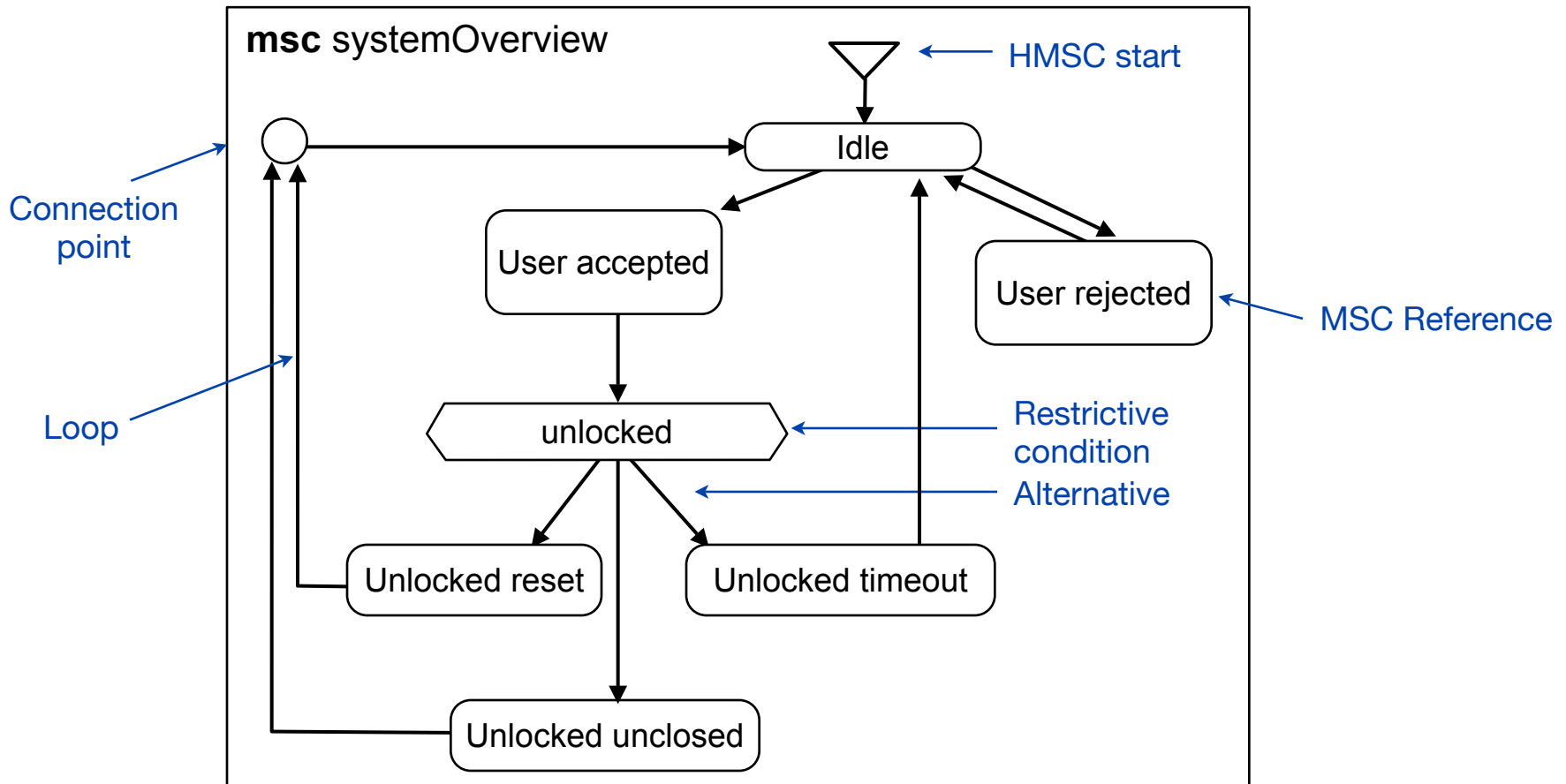


[Ø. Haugen: “MSC-2000 Interaction for the new Millenium”, sdl-forum.org/MSC2000present]

Method calls and control flow



High level MSCs

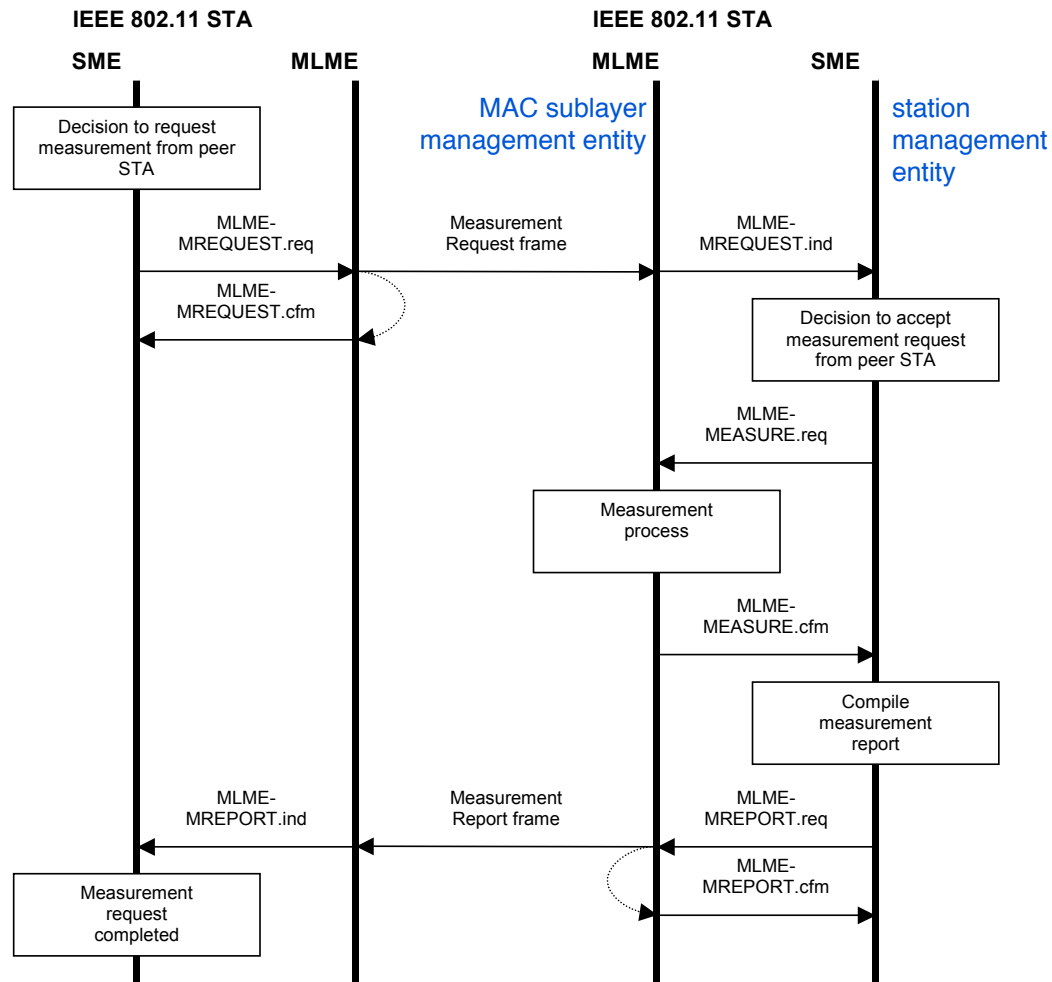


[Ø. Haugen: "MSC-2000 Interaction for the new Millenium", sdl-forum.org/MSC2000present]

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



Msg. sequence of a channel measurement

[IEEE Std. 802.11-2007]

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

[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]

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.)

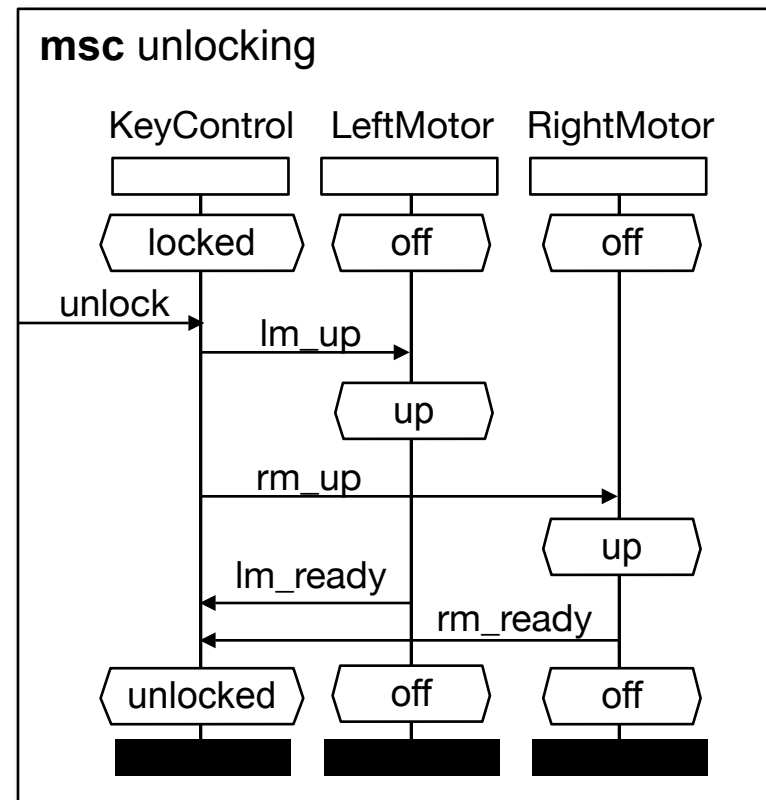
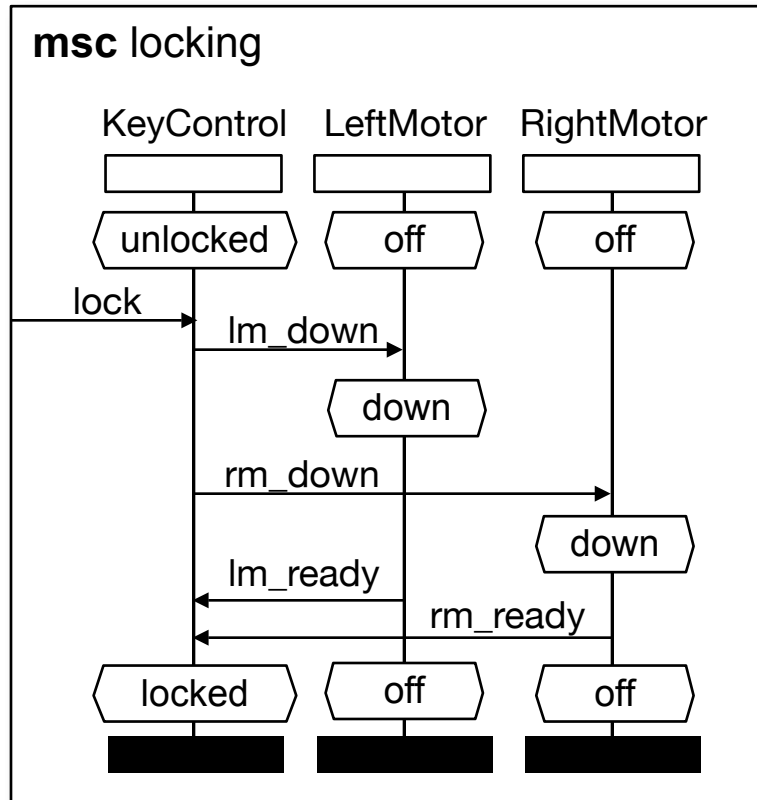
[I. Krüger, R. Grosu, P. Scholz, and M. Broy, “From MSCs to Statecharts”, DIPES’98]

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]

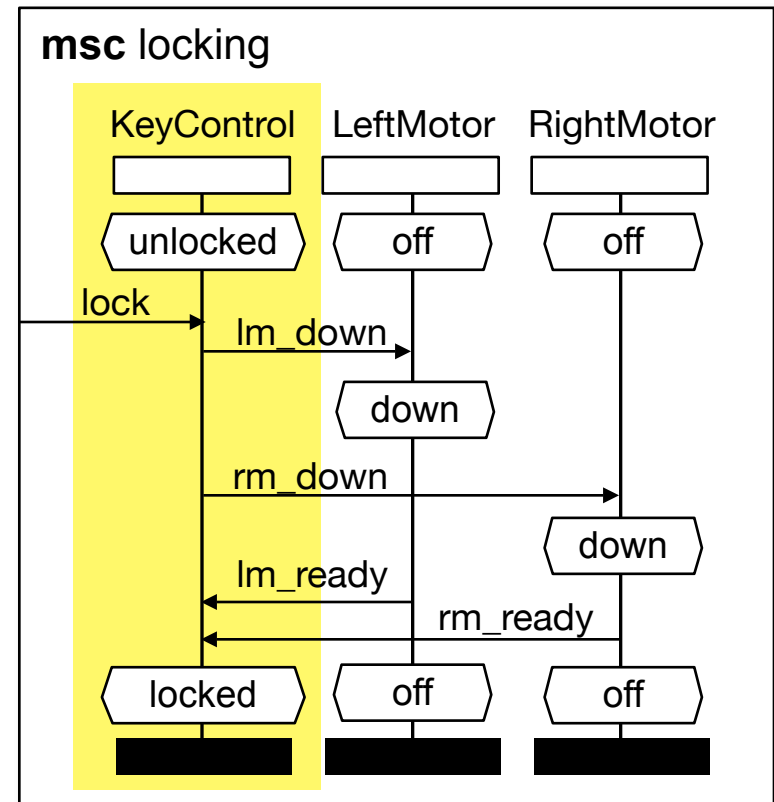
Example MSCs



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

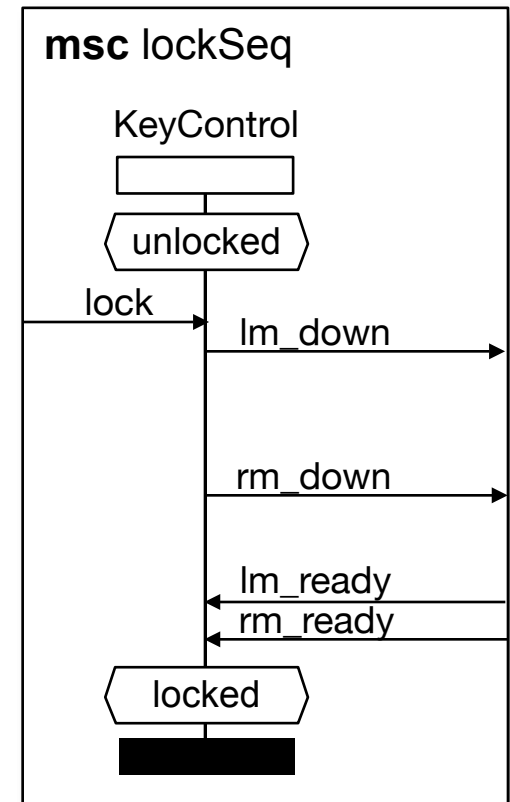
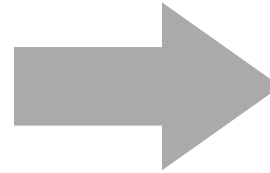
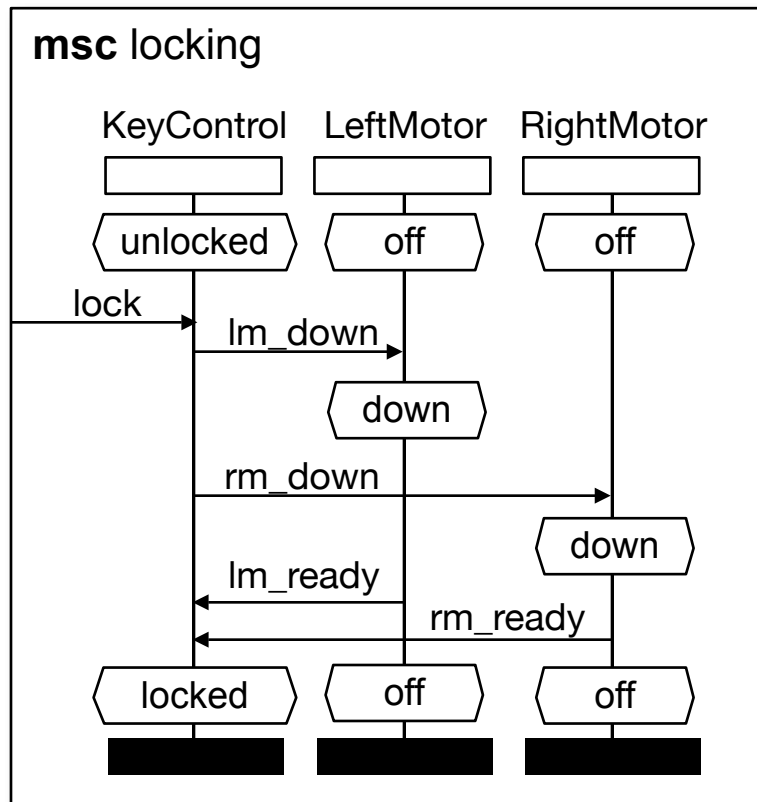
Projection

- ▶ 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]

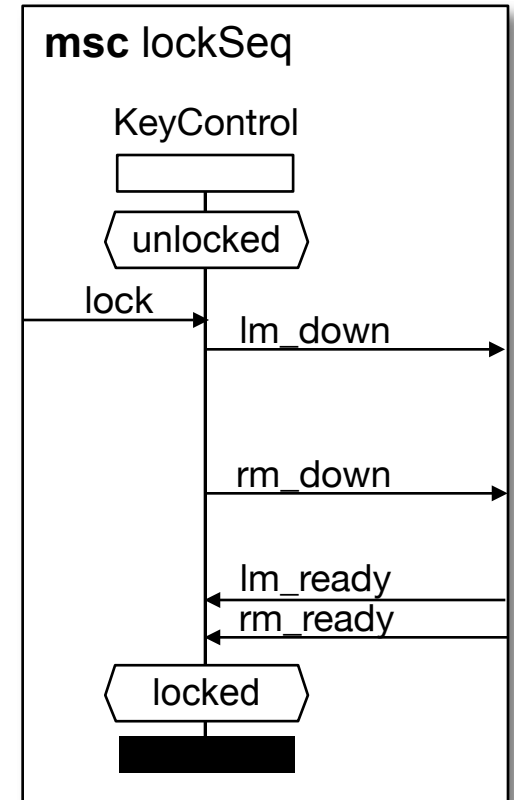
Projection



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

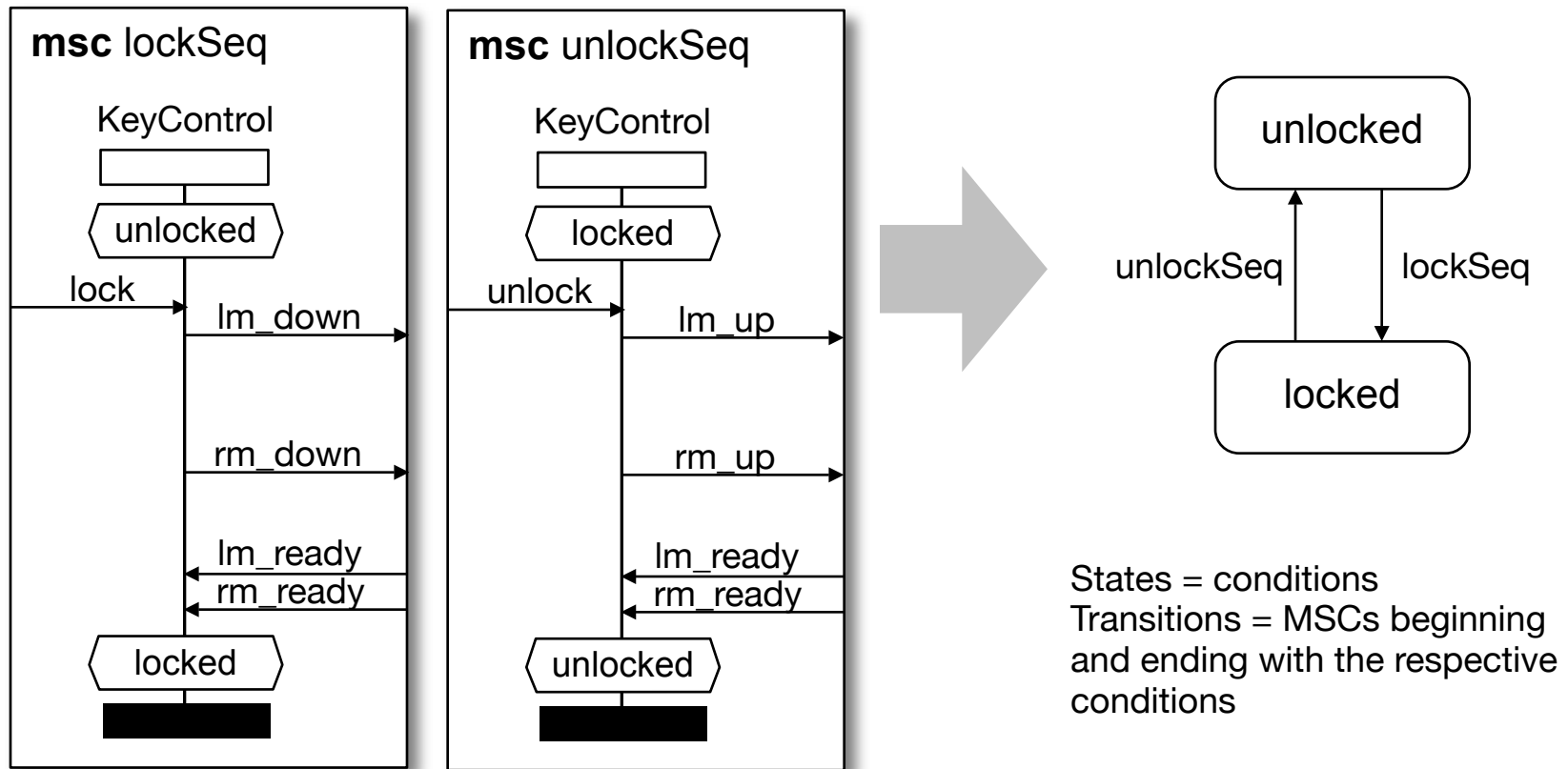
Normalization

- ▶ 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)



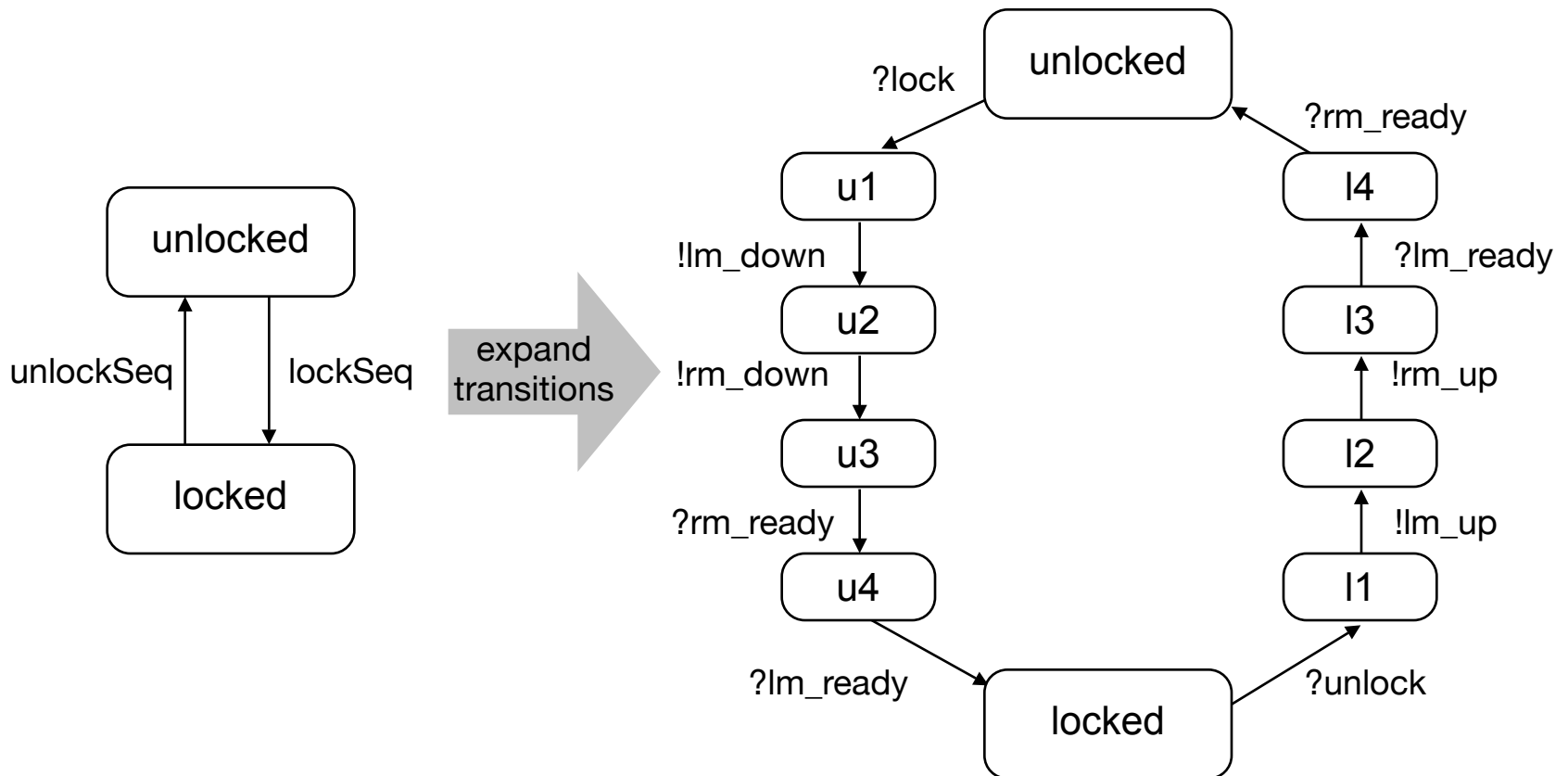
[I. Krüger, R. Grosu, P. Scholz, and M. Broy, "From MSCs to Statecharts", DIPES'98]

Transformation into MSC Automaton



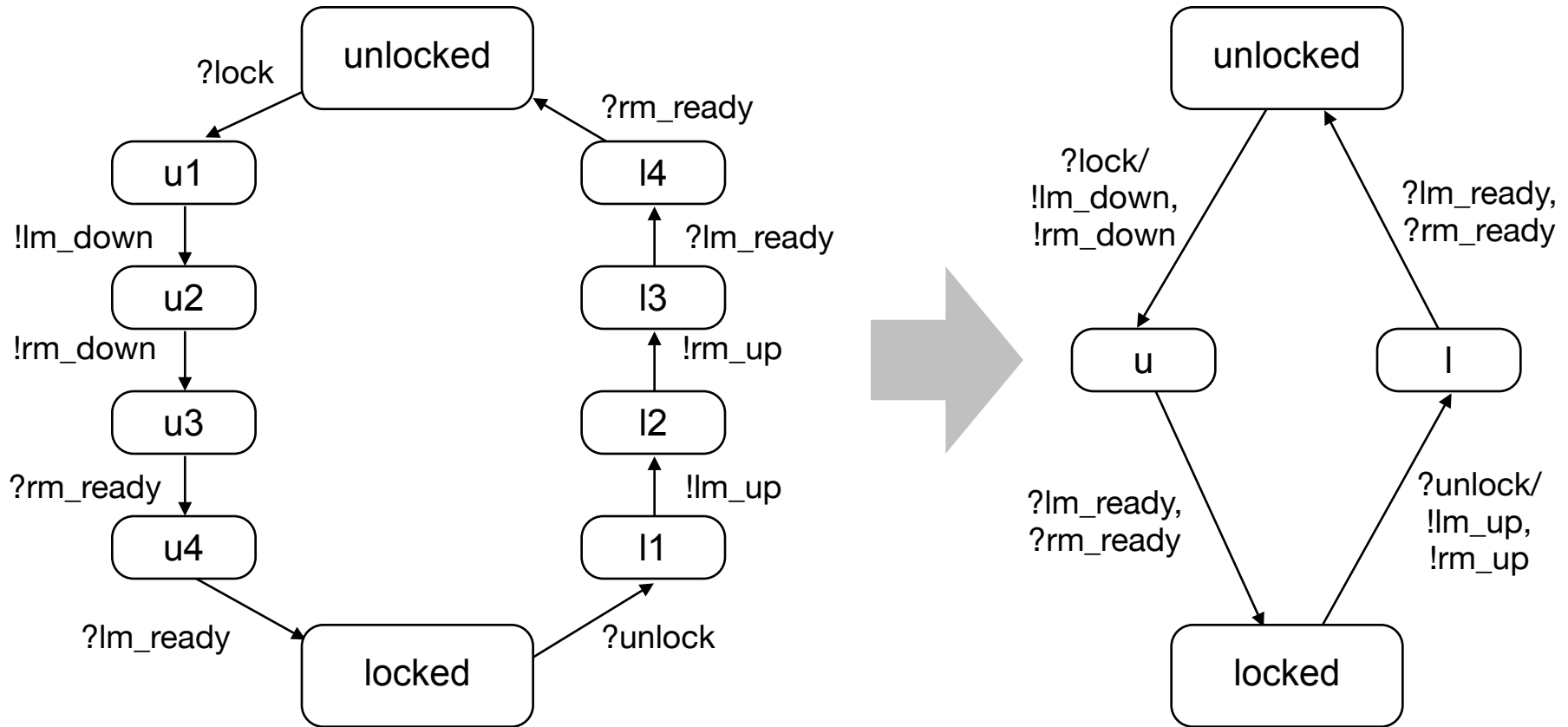
[I. Krüger, R. Grosu, P. Scholz, and M. Broy, "From MSCs to Statecharts", DIPES'98]

Transformation into CFSM



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

Optimization



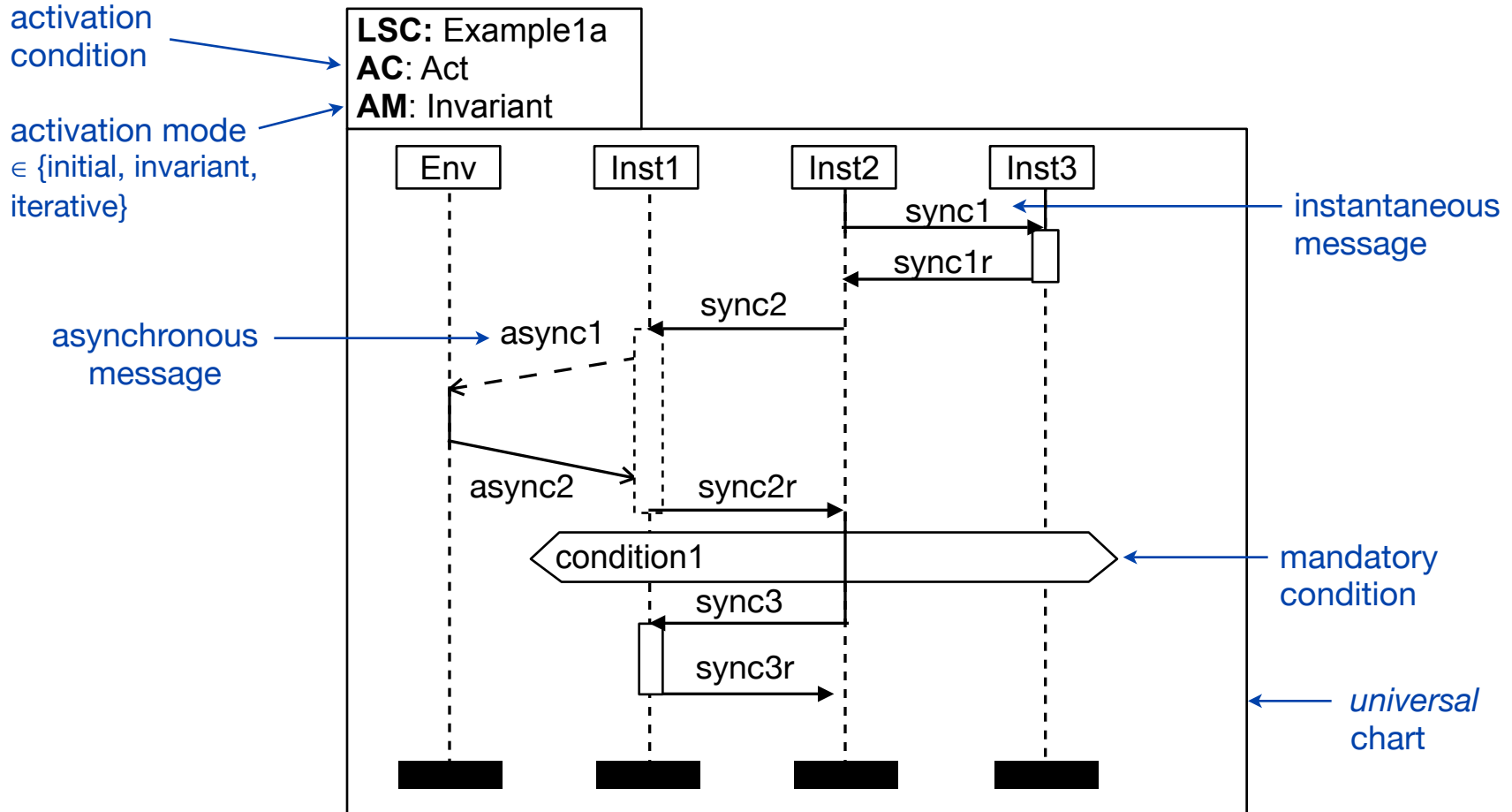
[I. Krüger, R. Grosu, P. Scholz, and M. Broy, "From MSCs to Statecharts", DIPES'98]

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

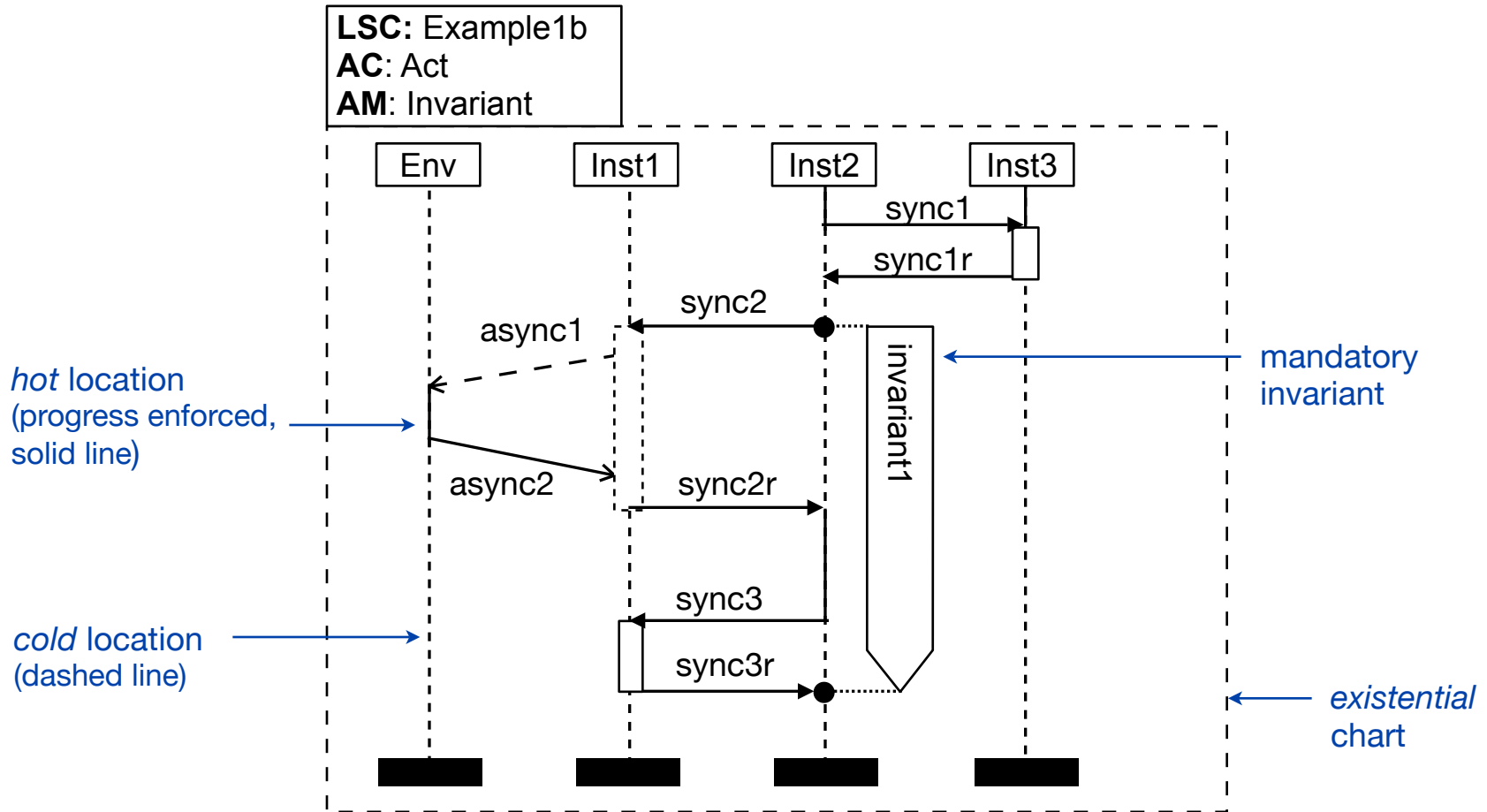
[W. Damm, D. Harel: “LSCs: Breathing Life into Message Sequence Charts”,
Formal Methods in System Design, 19, 45–80, 2001]

LSC Basics (1)



[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]

LSC Basics (2)



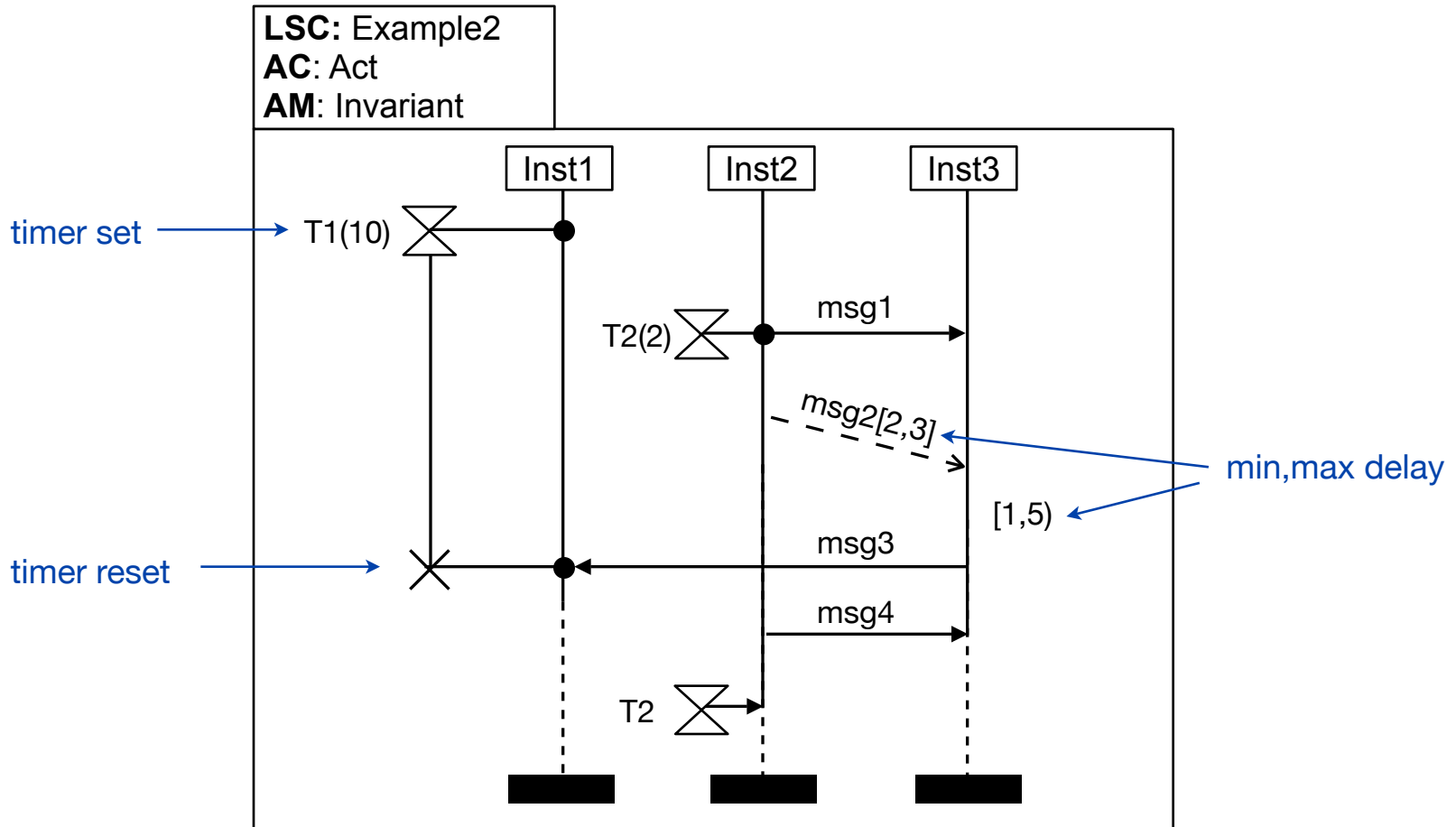
[M. Brill, W. Damm, J. Klose, B. Westphal, H. Wittke: Live Sequence Charts: An Introduction to Lines, Arrows, and Strange Boxes in the Context of Formal Verification. SoftSpez Final Report 2004]

Safety and Liveness in LSCs

	Mandatory (notation: solid line)	Provisional (notation: dashed line)
Chart	universal must be fulfilled by all runs	existential describes a possible run
Locations	hot progress is enforced	cold staying without progress is allowed
Messages	hot has to be delivered	cold may be lost
Conditions	hot must hold, otherwise abort	cold exit current chart if not met

[W. Damm, D. Harel: “LSCs: Breathing Life into Message Sequence Charts”,
Formal Methods in System Design, 19, 45–80, 2001]

Timing Constraints



[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

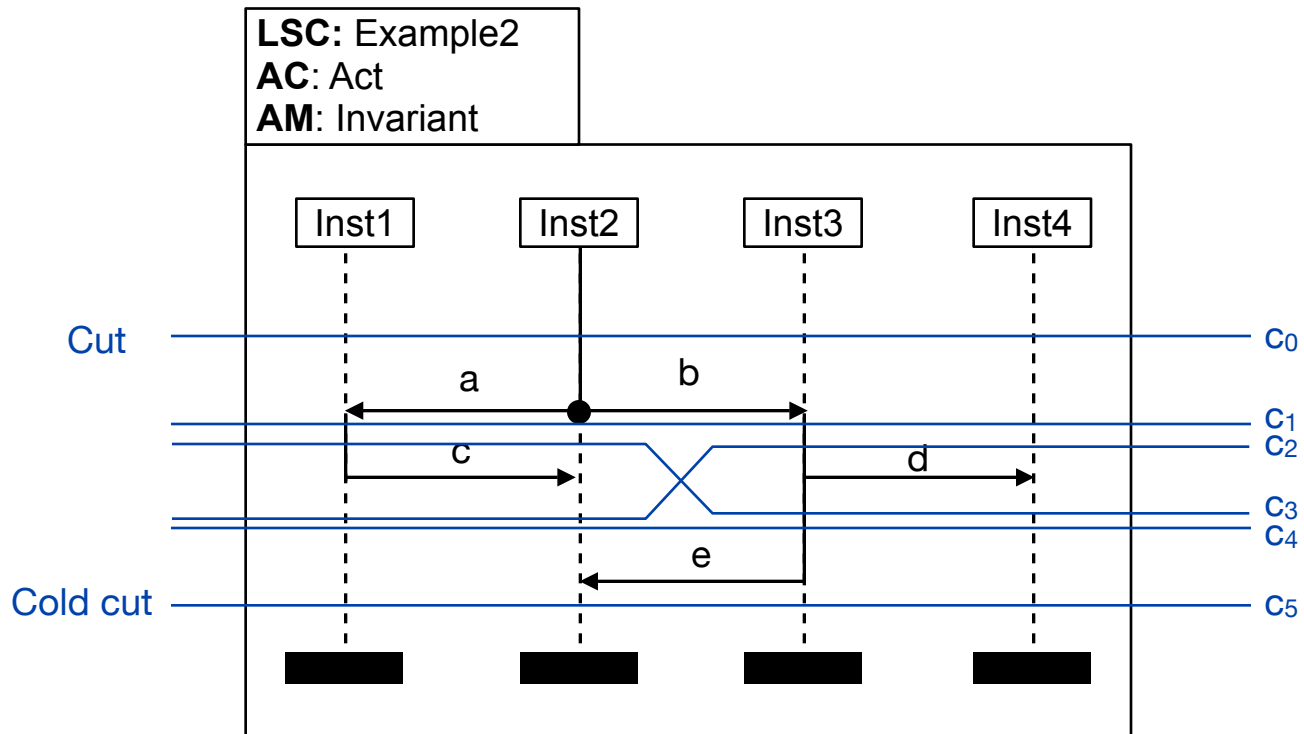
- ▶ Transformation of a LSC into a symbolic timed automaton that describes valid message sequences (*unwinding*)
- ▶ Basis for formal verification and validation
- ▶ Automaton model: Symbolic timed automaton
 - accepts infinite words (based on Büchi automata)
 - timed words: time is associated to symbols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

Unwinding (2)

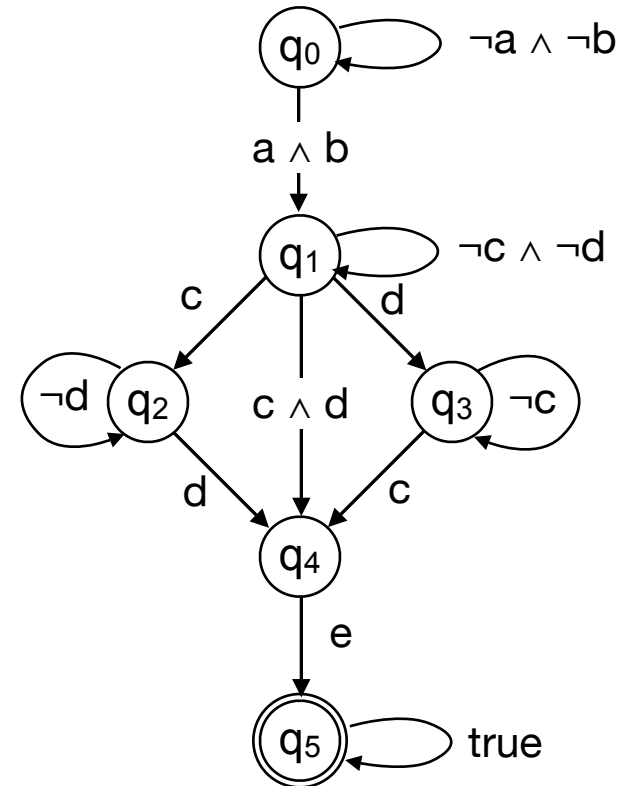
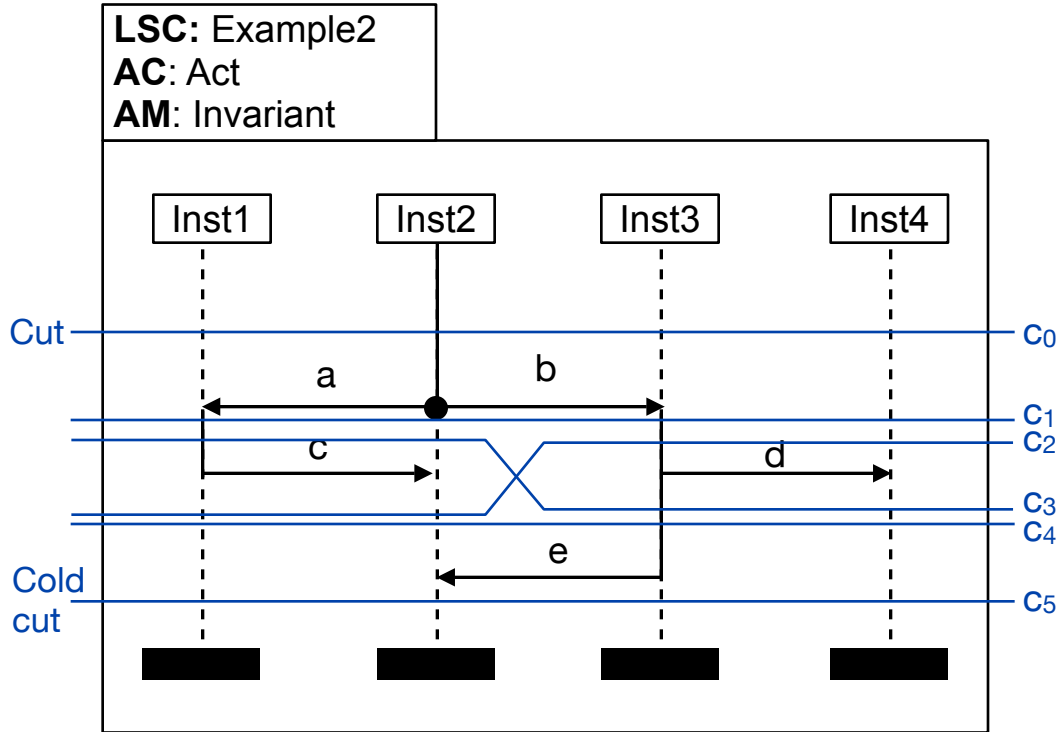
Moving a cut from top to bottom through the LSC:



Unwinding (3)

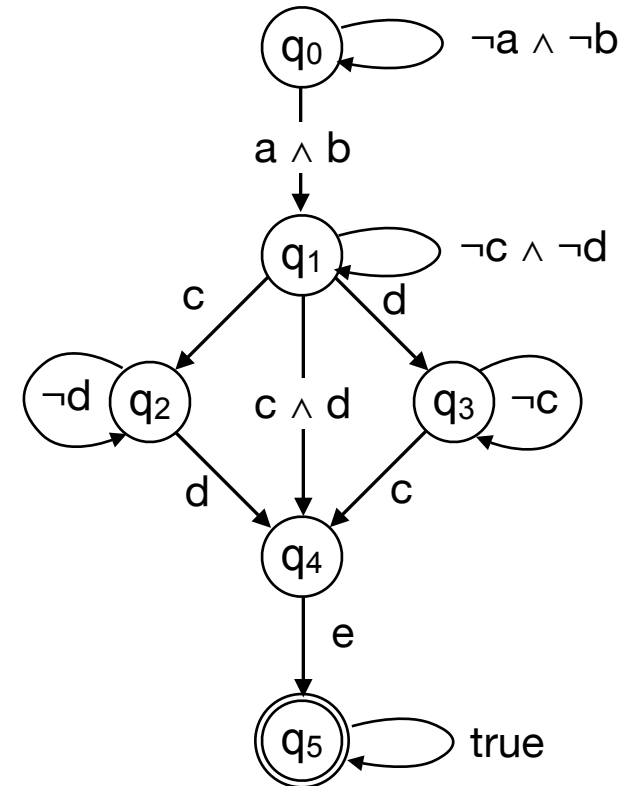
- ▶ Cuts become states in the automaton
- ▶ Transitions represent the successor relation among the cuts (successor relation reflects the partial order)
- ▶ Cold cuts become the acceptance states

Example (1)

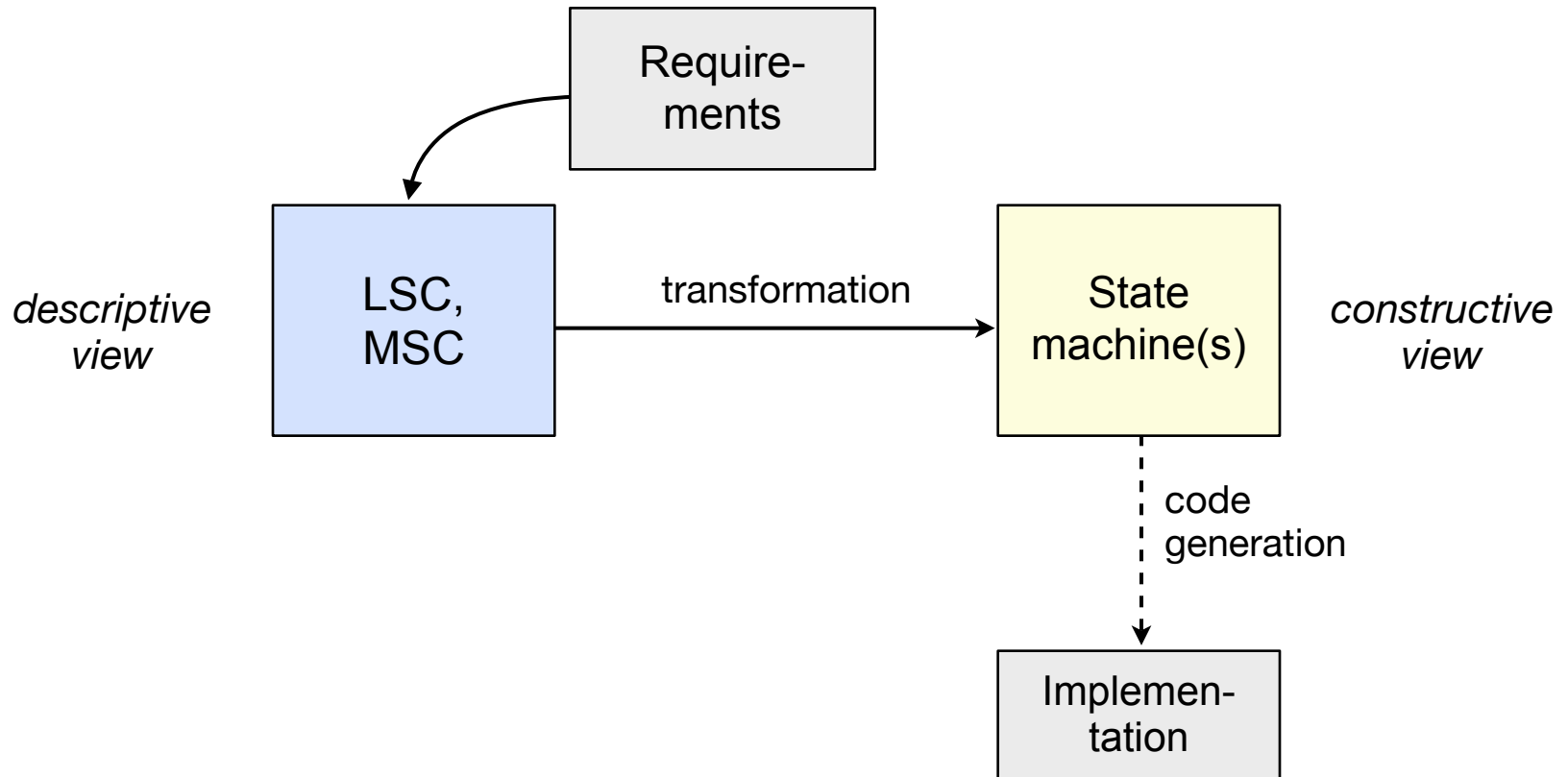


Example (2)

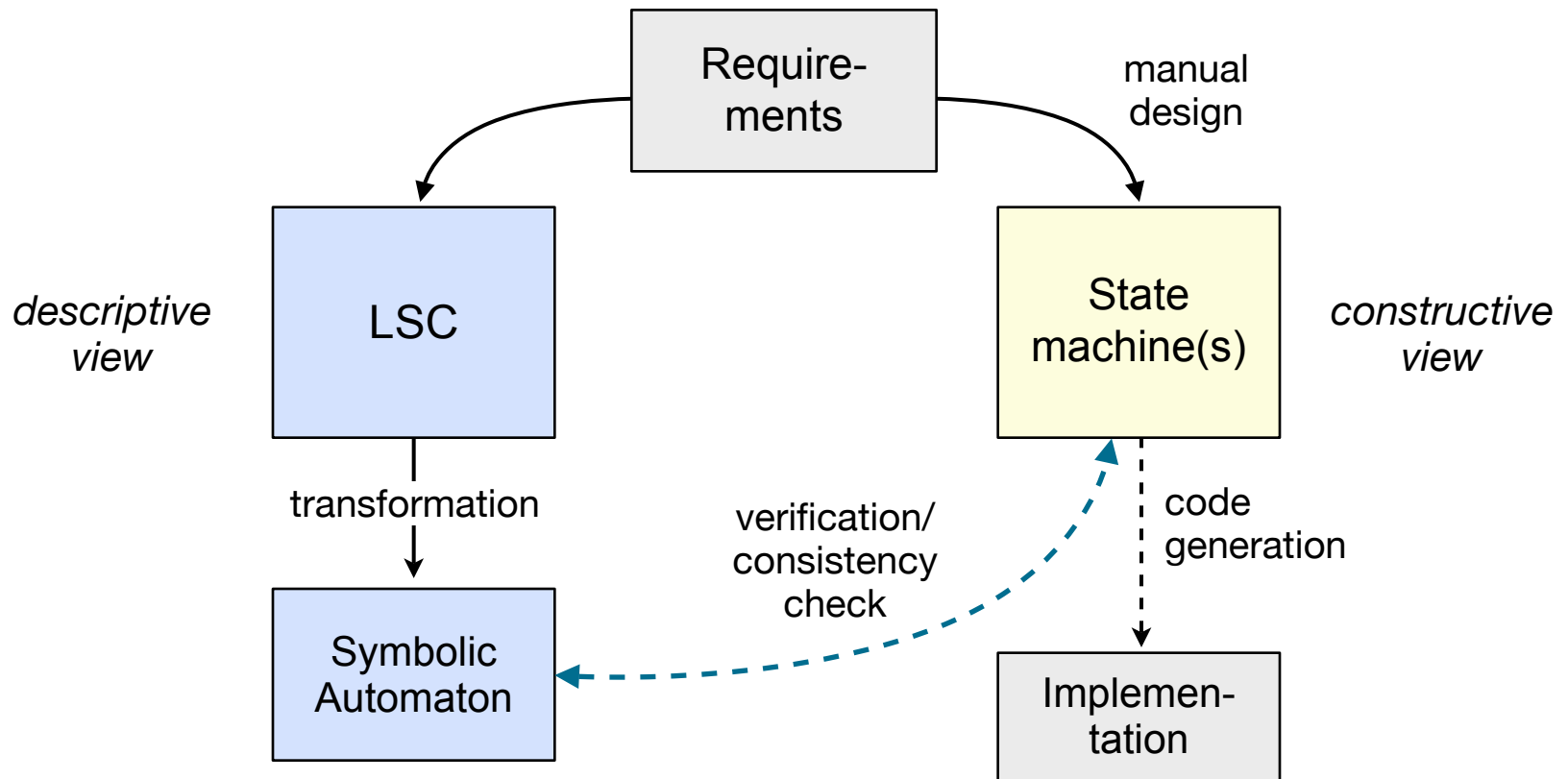
- ▶ Language of the automaton describes valid message sequences: $\{(a,b,d,c,e), (a,b,c,d,e), \dots\}$
- ▶ Note, that this is not a state machine as an implementation model



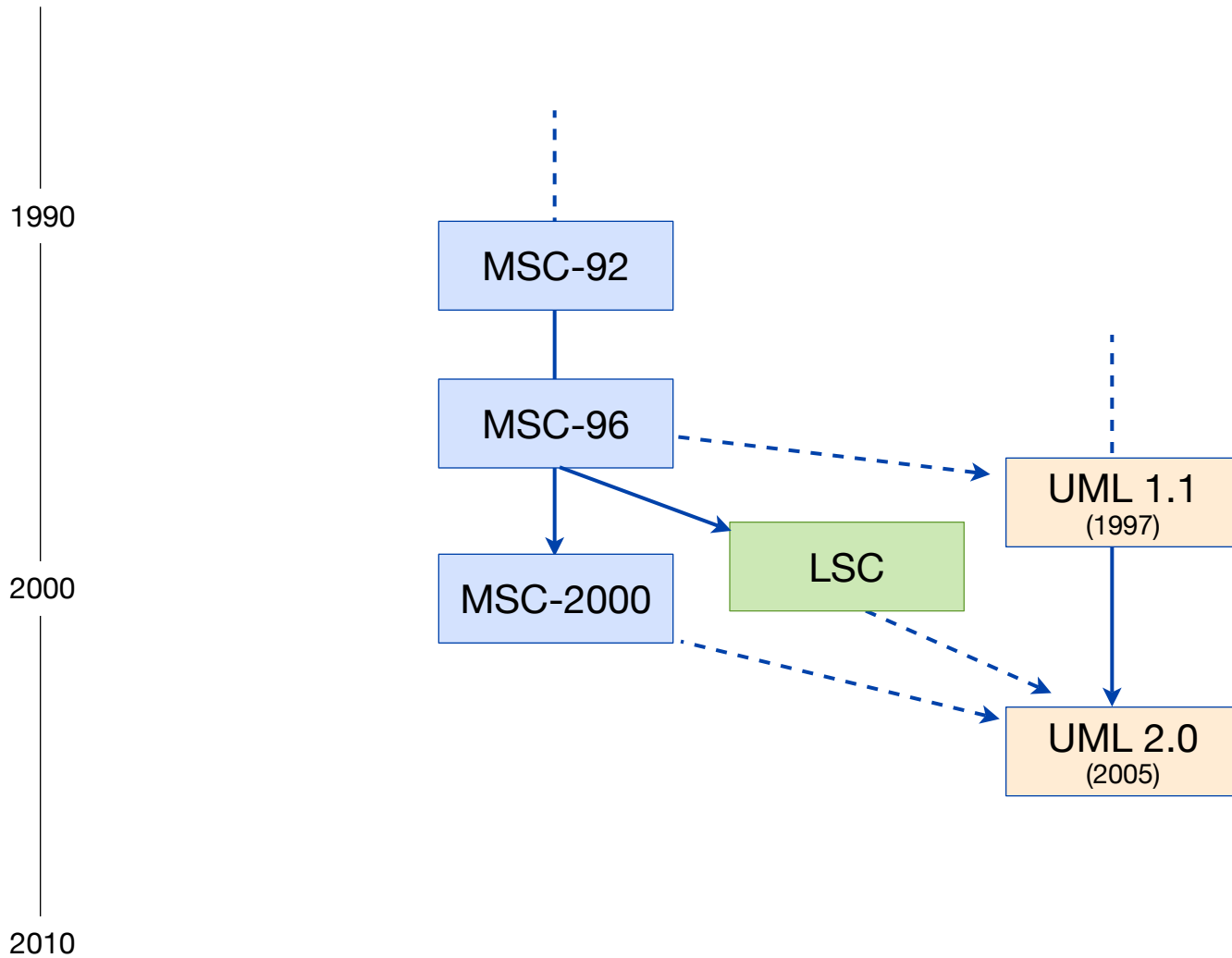
Code generation and Verification



Code generation and Verification



MSCs, LSCs, and UML



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