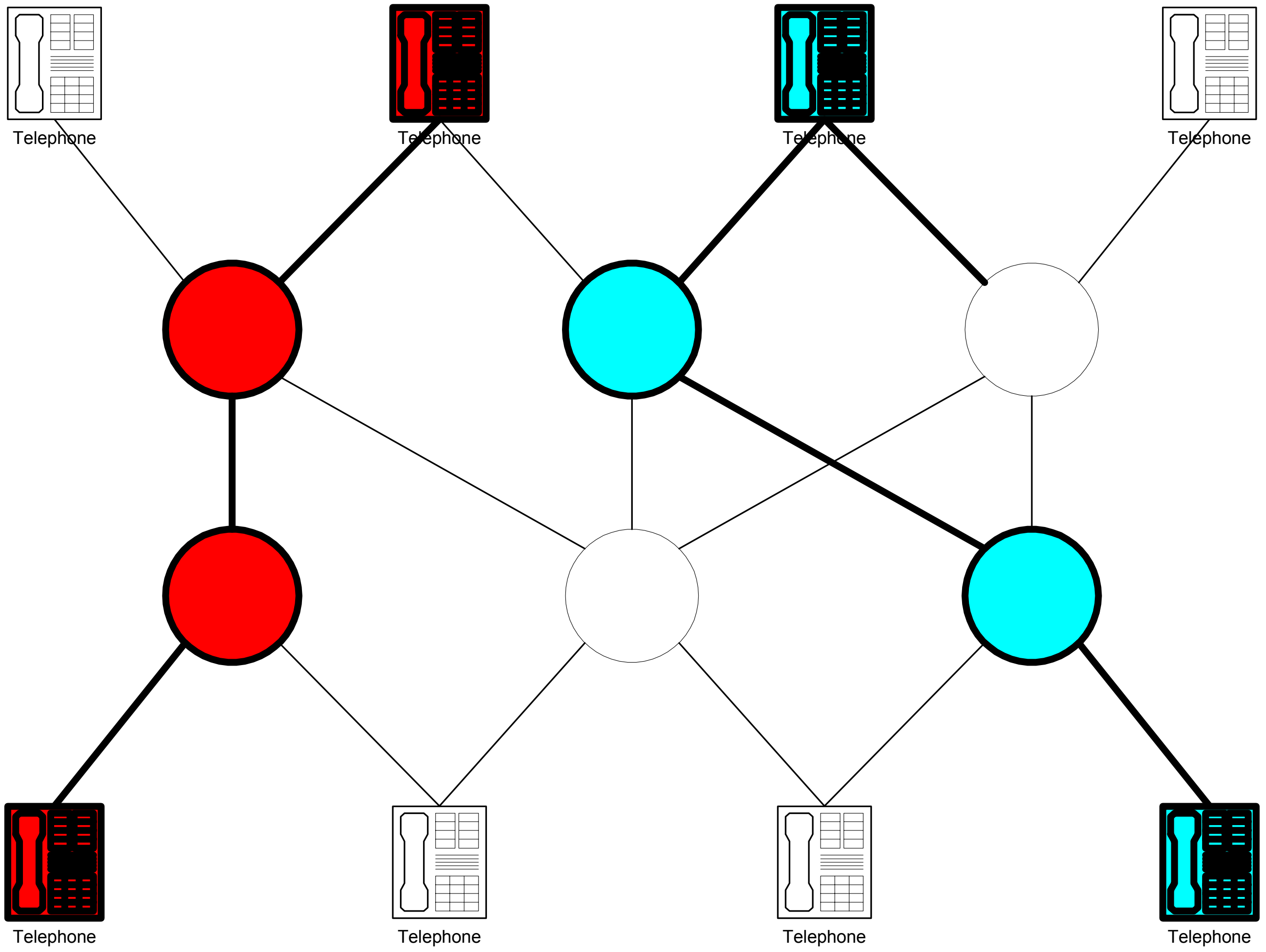
SDL

Characterization

- Specification language for the telecommunication domain
- Primarily process-oriented, even though data-oriented parts are also included
- Primarily targeted at the specification of processes, process states, and communication behavior
- Model-oriented specification, based on extended state machines
- Operational specification, i.e., SDL specifications are executable
- SDL is actually used in industry (e.g., at Ericsson)

Introduction

Application domain: telecommunication



Call processing

- Establish connection:
  - » A path through the network is established from the caller to the callee
  - » For the connection, processes are created dynamically on the switching computers and the terminal devices
- Conversation:
  - » Transfer of data along the connection
- Release connection:
  - » Processes created for the connection are destroyed

History of SDL (Specification and Description Language)

1972	Start of the development of SDL
1976	First version, issued by CCITT (Comité Consultatif International Télégraphique et Téléphonique), now ITU (International Telecommunications Union)
1980	Structuring concepts (hierarchy)
1984	Abstract data types
1988	Formal definition von SDL (FDT = Formal Description Technique)
1992	Object orientation, non-determinism, RPCs
1996	**„Smoothening“ of SDL-92 (SDL-96 is basis of this lecture)**
2000	Agents, interfaces, exception handling, composite states

Basic concepts of SDL

- A **system** consists of a set of communicating processes
- **Processes** may be created and destroyed dynamically
- Processes communicate by sending and receiving of **signals**
- Actual communication paths between processes are determined only at runtime
- The behavior of a process is defined by an **extended state machine** (explicit states + local variables)

Specification level and runtime level

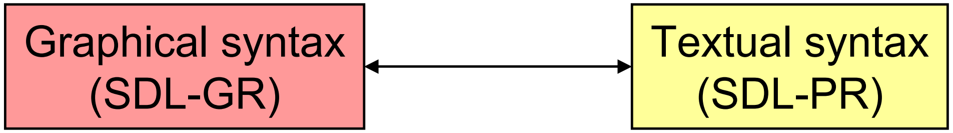
Specification level

- An SDL specification is structured hierarchically:
  - » **System**: root
  - » **Block**: inner node (as many nesting levels as desired)
  - » **Process**: leaf
- Note: A leaf of the hierarchy represents a **set** of process instances sharing the same specification
- Blocks are introduced to structure the specification
- **Block** and **process types** allow reuse at multiple places in the hierarchy

Runtime level

- At runtime, a system consists of a dynamic set of **process instances**
- This set is flat, i.e., some process *Q* generated by some process *P* exists independently of *P*
- The communication paths between process instances are determined dynamically
- In particular, processes *P* and *Q* may communicate even if they are not connected by a static communication path

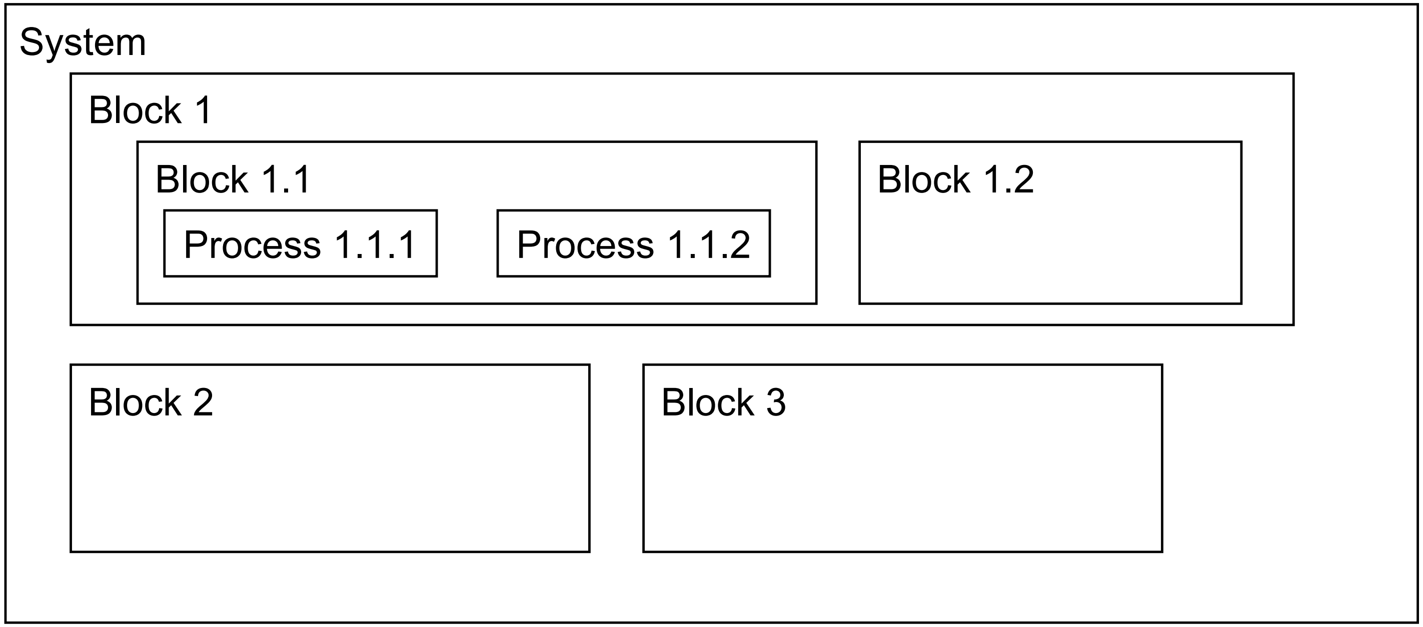
Graphical and textual syntax



- SDL specifications may be written alternatively in graphical or textual syntax
- Both notations are equivalent and may be transformed into each other
- In particular, the graphical notation is complete
- In the sequel, the graphical notation will be preferred

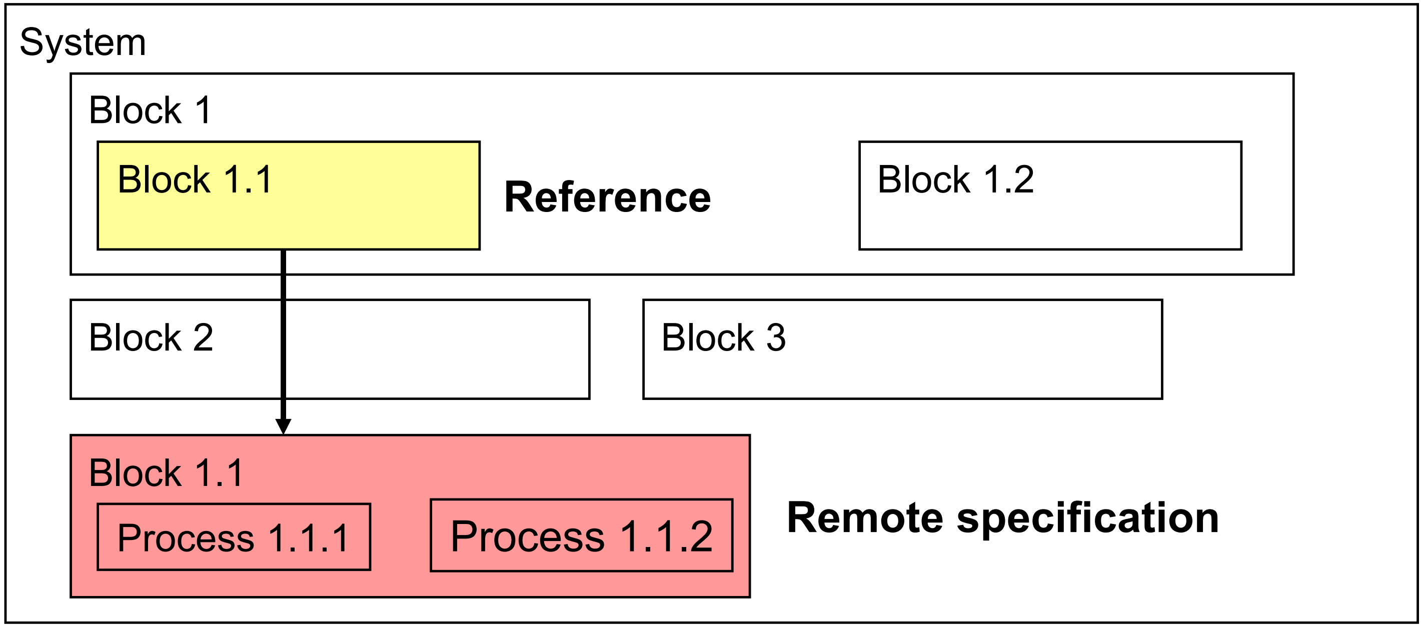
Basic SDL

Structure of specifications (1): hierarchies



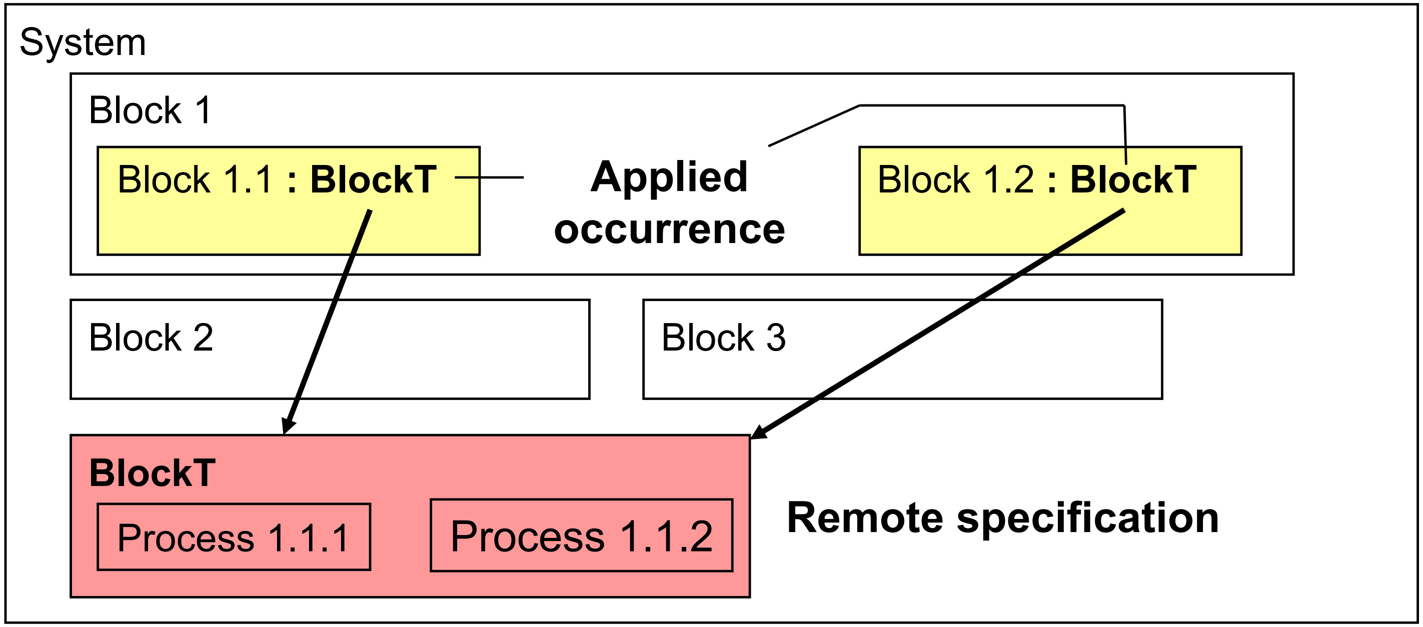
- Specifications of systems, blocks, and processes may be nested physically (like e.g. nested procedures in Pascal)
- Disadvantages:
  - Mixing of hierarchy levels
  - Unreadable in case of large specifications

Structure of specifications (2): remote specifications



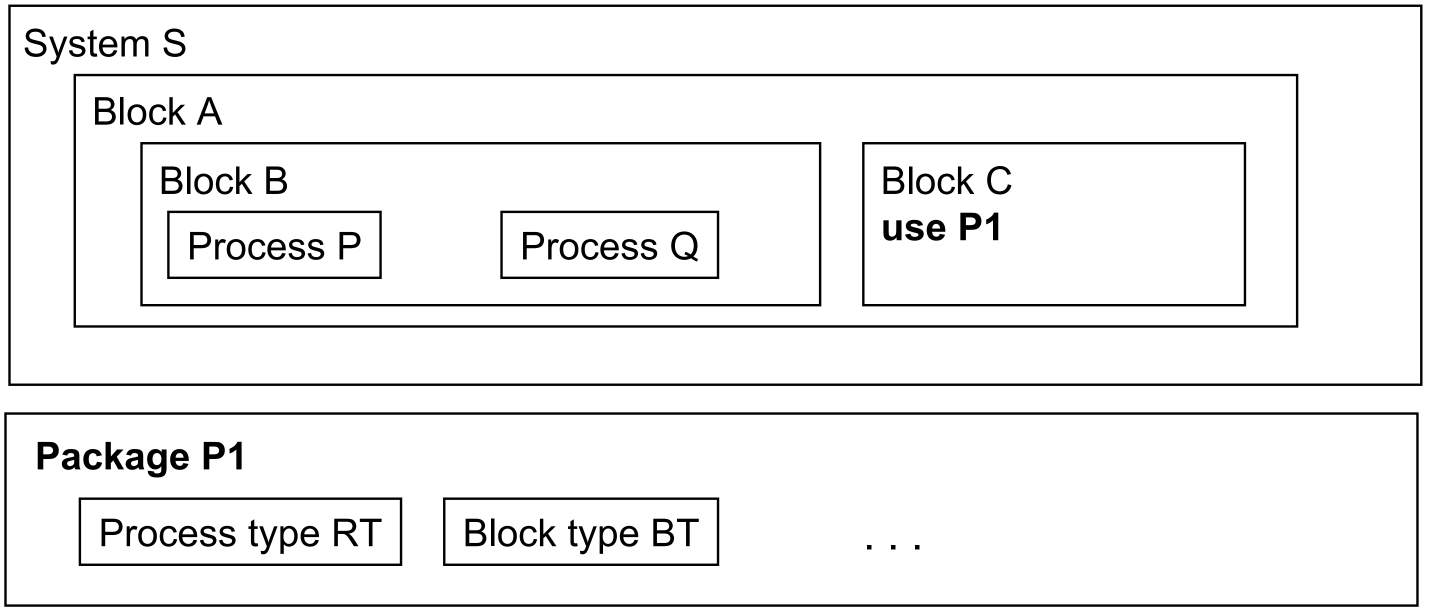
- Instead of physical nesting, a **reference** to a **remote specification** may be introduced (cf. e.g. local packages in Ada)
- Semantics: replace the reference with the specification
- Disadvantages:
  - No reuse (exactly one reference)
  - Remote specification depends on the context of the reference

Structure of specifications (3): block and process types



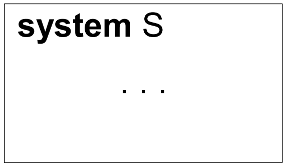
- In case of a **block** or **process types**, specifications may be reused at different locations in the hierarchy
- Type declarations do not depend on the context of the applied occurrence

Scopes and packages

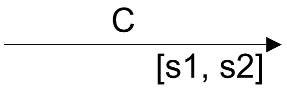


- Systems, blocks, and processes define scopes
- **Scoping rules** are the same as in Pascal (i.e., a declaration is visible in its scope and all enclosing scopes unless hidden)
- Reusable type declarations are provided in **packages**
- **Import clauses** make declarations contained in packages visible elsewhere

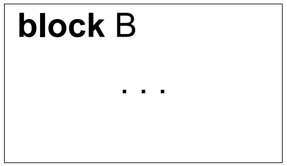
Block and process interaction diagrams



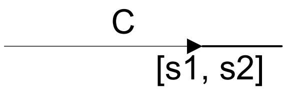
Block interaction diagram for System S



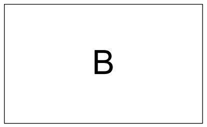
Uni-directional channel C (without delay) for transmitting signals s1 and s2



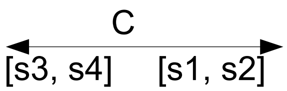
Block or process interaction diagram for block B



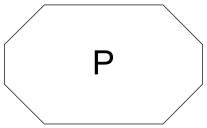
Uni-directional channel C (with delay) for transmitting signals s1 and s2



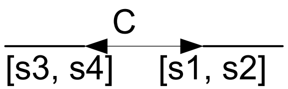
Reference to remote specification of B



Bi-directional channel C (without delay) for transmitting signals s1, s2 and s3, s4, respectively



Reference to remote specification of P

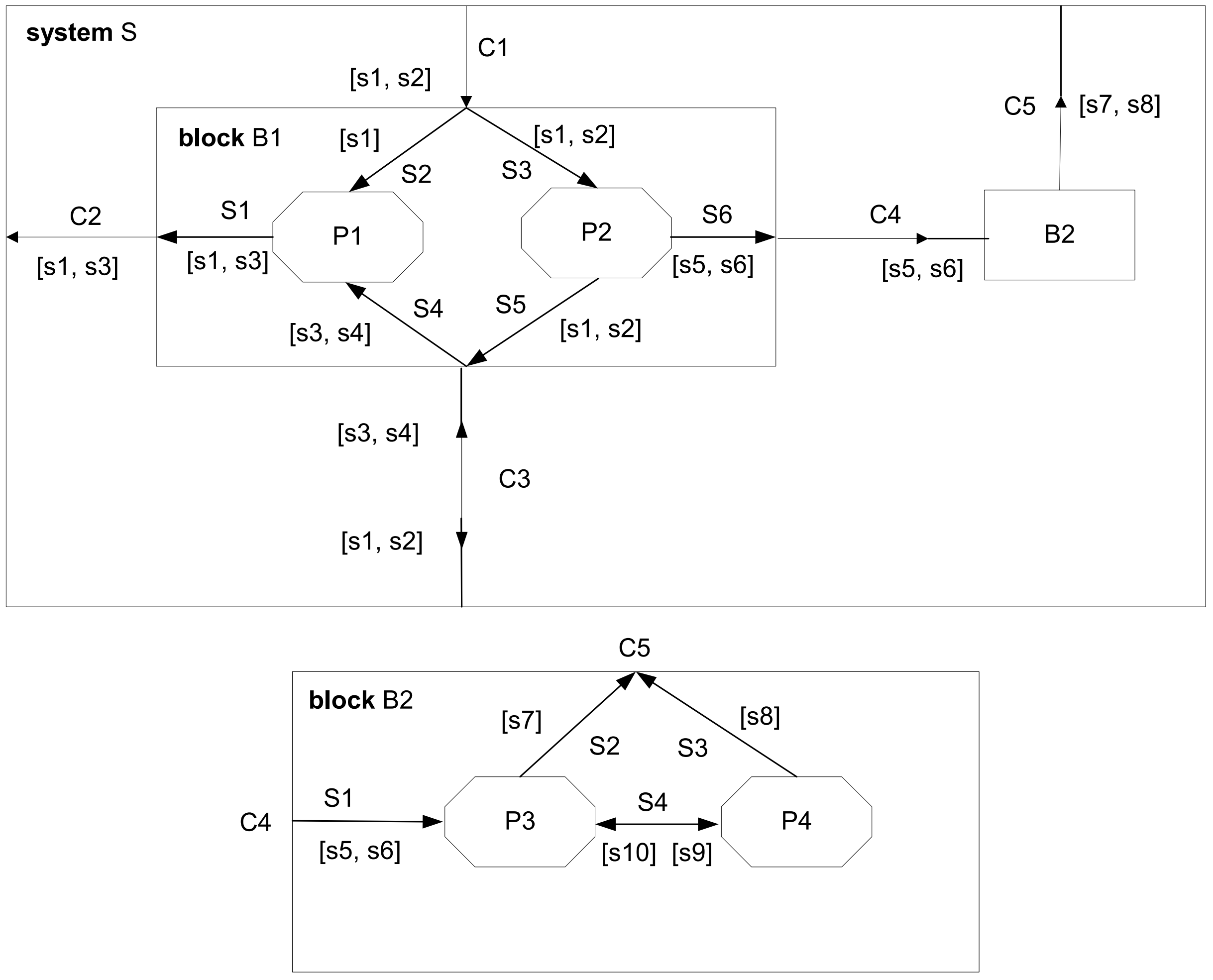


Bi-directional channel C (with delay) for transmitting signals s1, s2 and s3, s4, respectively

Channels and signal routes

- Communication paths between blocks are called **channels**
- Communication paths between processes are called **signal routes**
- Signal routes are always undelayed
- Channels may or may not have a delay
- A bi-directional communication path stands for two uni-directional communication paths
- Association between communication paths in hierarchical specifications:
  - » In physically nested diagrams: graphical position
  - » In case of remote specifications: matching names
  - » m:n associations between parent and child are allowed
  - » Signal flow may even be non-deterministic (at least statically)
  - » Balancing rule for mutually associated communication paths:
    - ⇨ The set of transmitted signals must be the same in parent and child

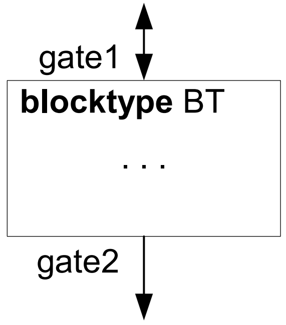
Example of an interaction diagram



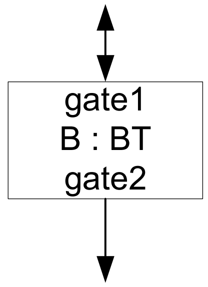
Process and block types

- Process and block types enable reuse in the specification
- Process and block types may be declared locally, or they may be imported from packages
- An applied occurrence of a block type stands for exactly one block
- An applied occurrence of a process type stands for a set of process instances
- The cardinality of process instances may be constrained as follows (the lower bound simultaneously defines the number of initial instances):
  - » (0,n): unconstrained number of instances (default)
  - » (1,n): at least one instance
  - » (1,1): exactly one instance
  - » (0,1): at most one instance
- **Gates** define the connection points of process and block types (roughly corresponding to formal parameters of procedures)

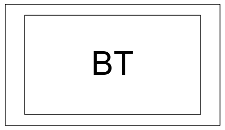
Graphical notation of block and process types



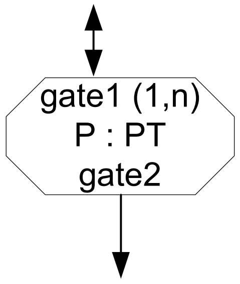
Interaction diagram for block type BT



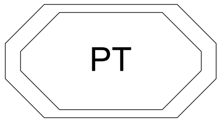
Instance B of block type BT with gates gate1 and gate2



Reference to the remote specification of block type BT



Instance P of process type PT with cardinality (1,n) and Gates gate1 and gate2



Reference to the remote specification of process type PT

Definition of signals

- In the simplest case, a **signal** does not carry further information, e.g.,
  - » `signal Ready, Active, Suspended ...;`
- In general, a signal may carry a **tuple** of **values**, e.g.
  - » `signal State(TState);`
  - »

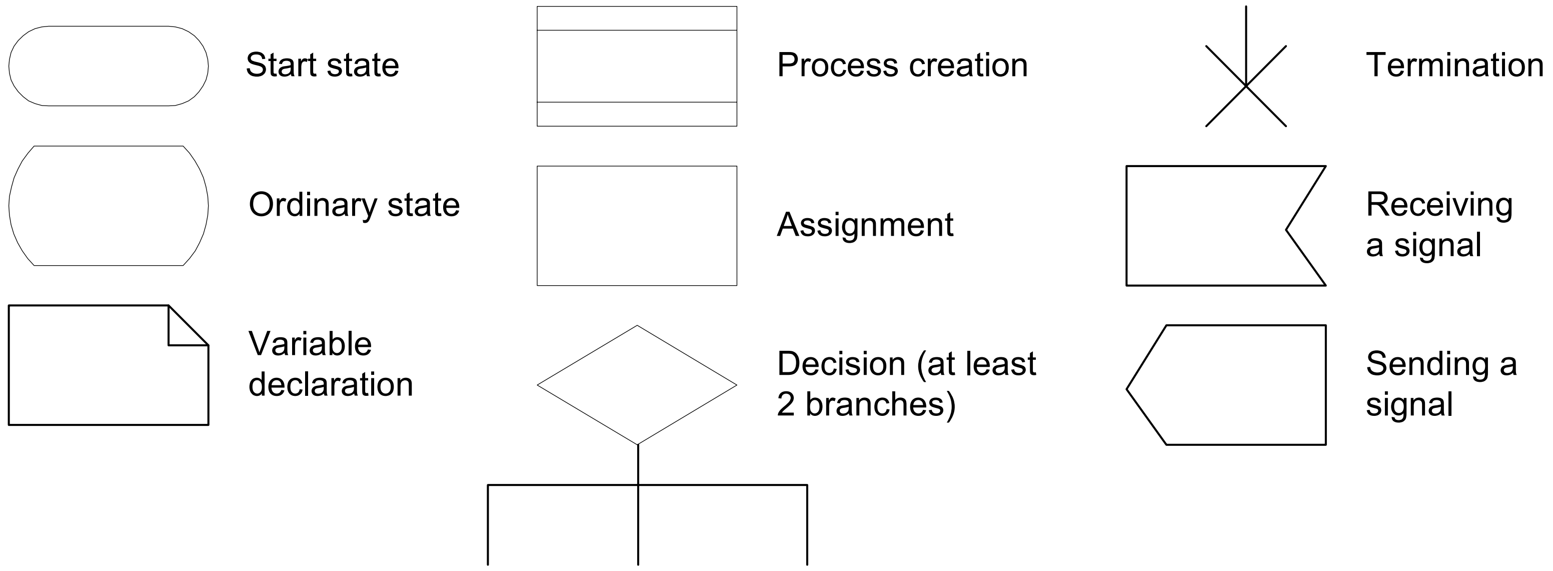
```
newtype TState
        literals Ready, Active, Suspended ...;
endnewtype TState; /* Aufzählungstyp */
```

- The following **types** are available:
  - » Pre-defined types (integer, real, boolean, character, charstring)
  - » Records
  - » Arrays
  - » Sets
  - » Enumeration types
  - » Abstract data types

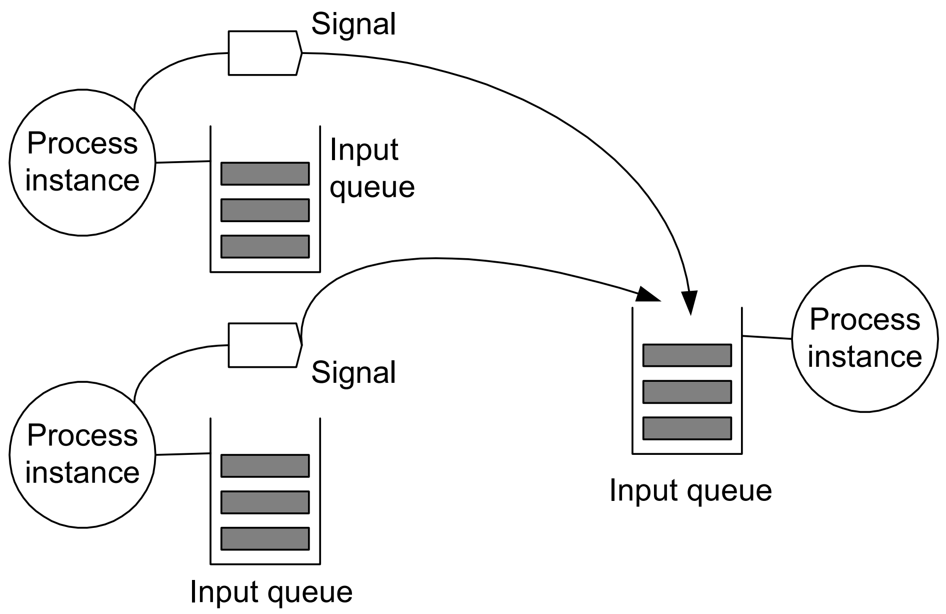
Extended state machines

- The behavior of processes is defined by **extended finite automata** (states, transitions, local variables)
- A process is idle while residing in a **state**
- An incoming signal triggers a **state transition**
- During a state transition, the following **actions** may be executed:
  - » Assignment of values to local variables
  - » Decisions (branches)
  - » Sending of signals
  - » Creation of process instances
  - » Termination (a process instance may terminate only itself)
- In the graphical notation, states may be replicated to improve readability

Graphical notation of state machines



Communication between process instances



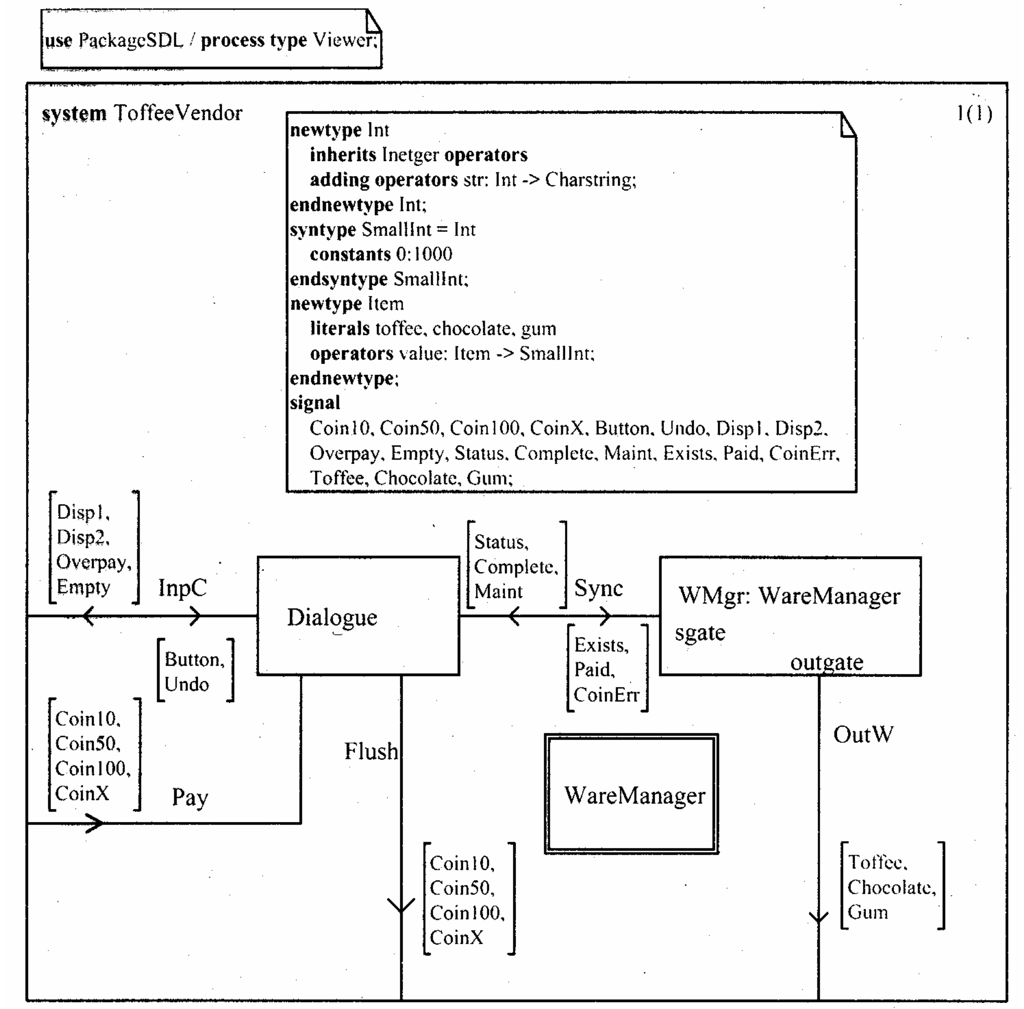
- Each process has an **input queue** of signals which are processed in FIFO order
- A transition **consumes** the head of the input queue
- Signals without a transitions are consumed without state change **(implicit transition)**
- Each signal is sent to only one receiver (no broadcasting or multicasting)

Addressing of processes

- Each process instance is designated by a unique **process identifier** (of type `PId`)
- Process identifiers may be sent as values of signals and may be stored in local variables ⇒ communication at runtime need not occur along the static channels and signal routes
- **Explicit addressing**: When sending a signal, the receiver is specified explicitly, using the following pre-defined variables if appropriate:
  - **`self`**: the current process
  - **`sender`**: the process from which the last signal was received
  - **`offspring`**: the last process created by the current process
  - **`parent`**: the process which created the current process
- **Implicit addressing:** Specification of a communication path along which the signal will be sent:
  - Signal route
  - Gate

Example

System `ToffeeVendor`

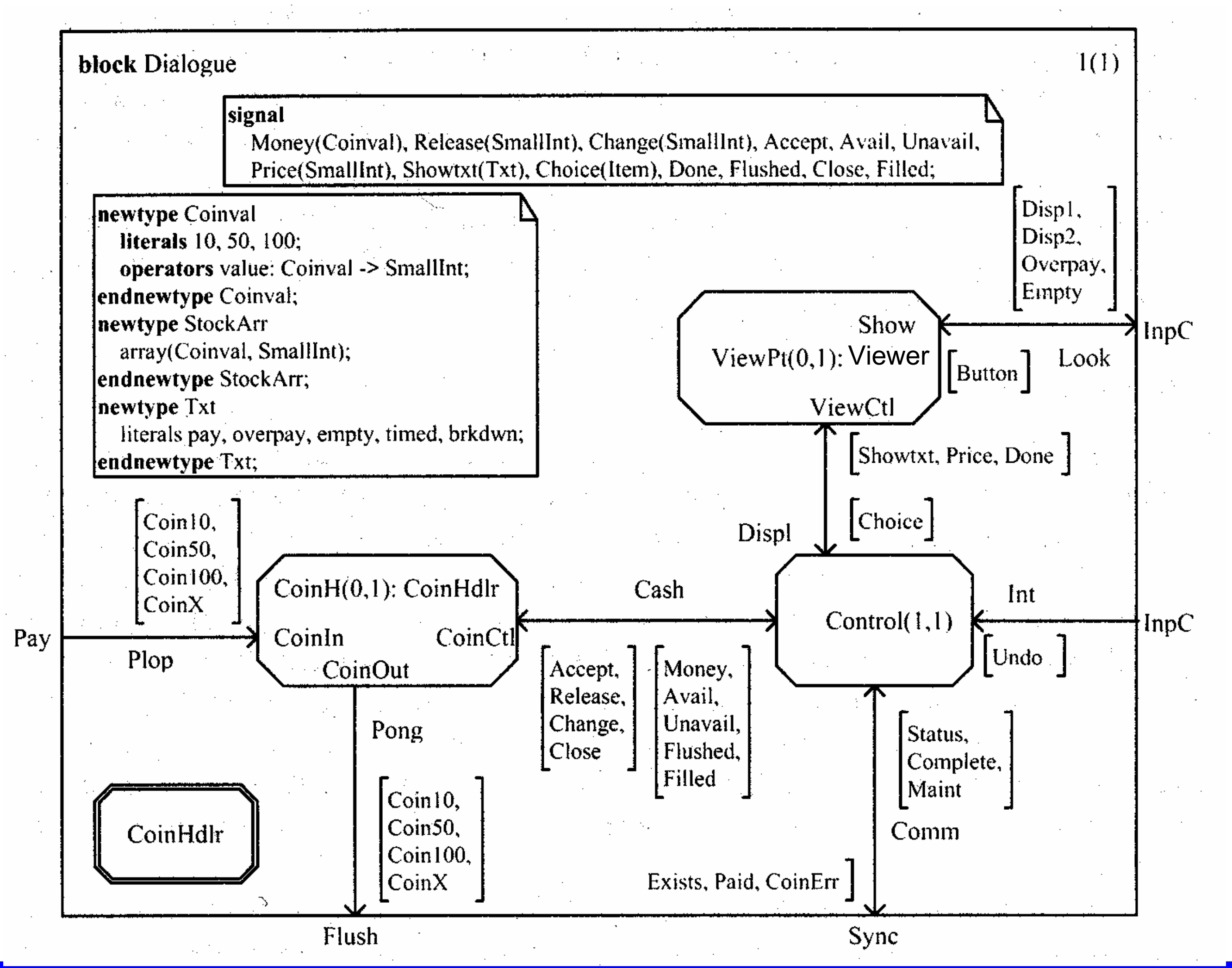


use PackageSDL / **process type** Viewer;

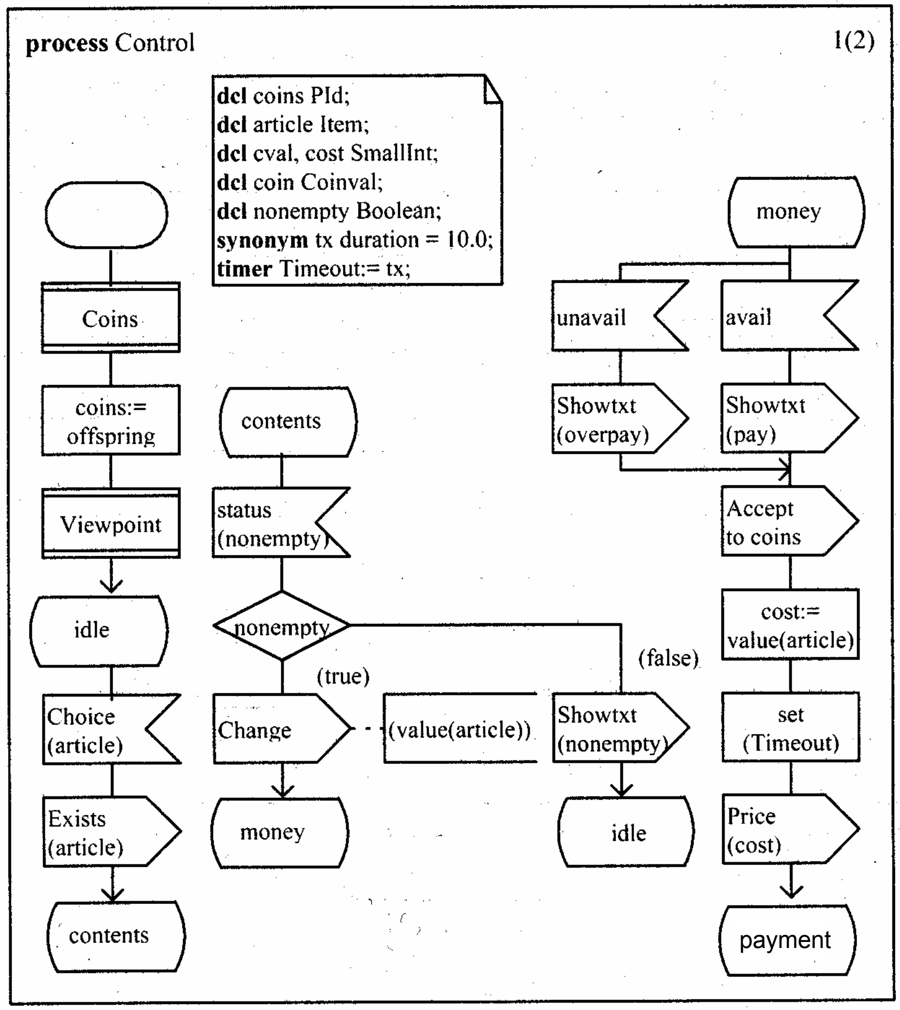
system ToffeeVendor 1(1)

```
newtype Int
   inherits Inetger operators
   adding operators str: Int -> Charstring;
endnewtype Int;
syntype SmallInt = Int
   constants 0:1000
endsyntype SmallInt;
newtype Item
   literals toffee, chocolate, gum
   operators value: Item -> SmallInt;
endnewtype;
signal
   Coin10, Coin50, Coin100, CoinX, Button, Undo, Disp1, Disp2,
   Overpay, Empty, Status, Complete, Maint, Exists, Paid, CoinErr,
   Toffee, Chocolate, Gum;
```

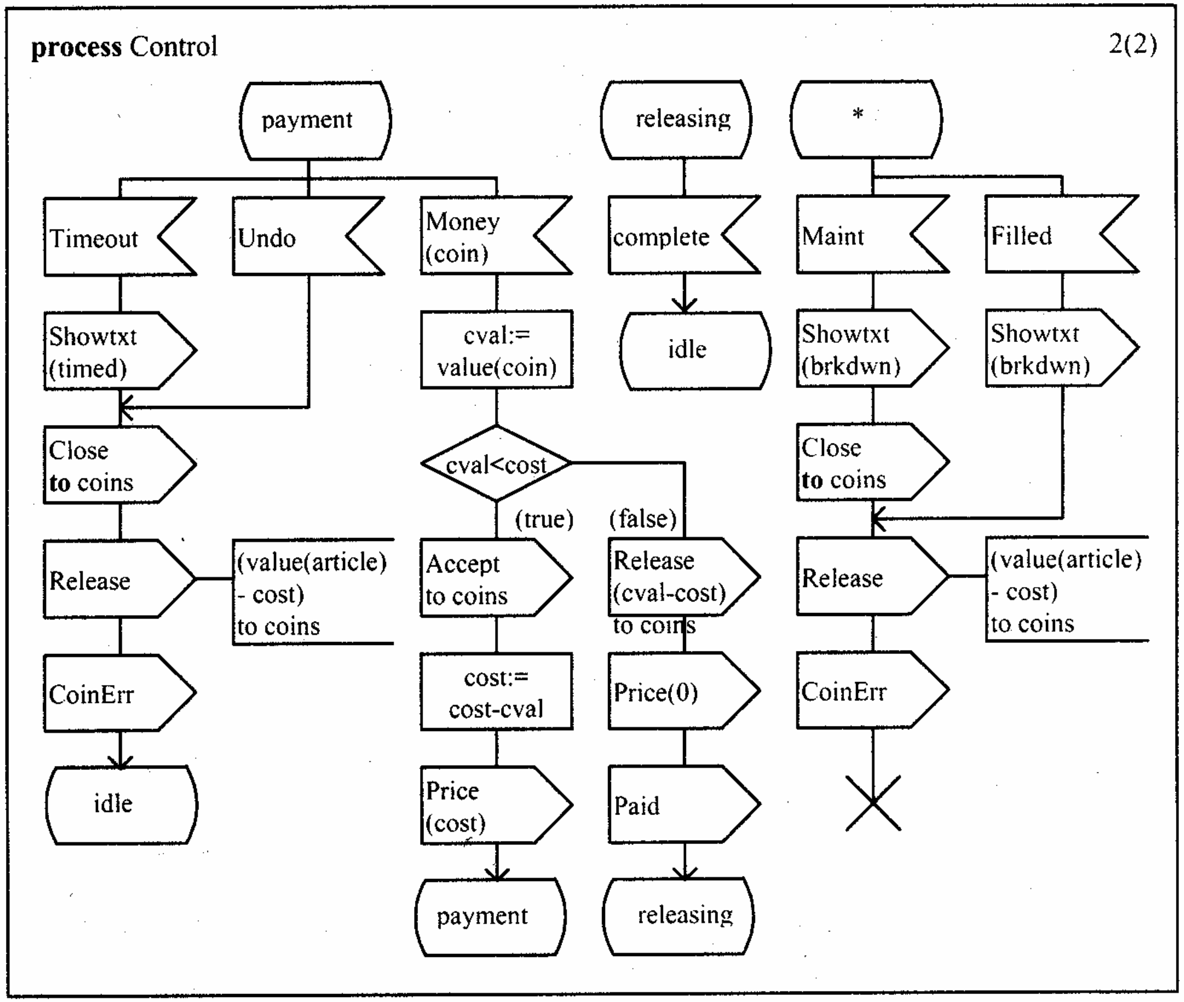
Block `Dialogue`


block Dialogue
1(1)
signal Money(Coinval), Release(SmallInt), Change(SmallInt), Accept, Avail, Unavail, Price(SmallInt), Showtxt(Txt), Choice(Item), Done, Flushed, Close, Filled;
newtype Coinval
literals 10, 50, 100;
operators value: Coinval -> SmallInt;
endnewtype Coinval;
newtype StockArr
array(Coinval, SmallInt);
endnewtype StockArr;
newtype Txt
literals pay, overpay, empty, timed, brkdwn;
endnewtype Txt;
[Disp1, Disp2, Overpay, Empty]
Show
ViewPt(0,1): Viewer
ViewCtl
[Button]
Look
InpC
[Showtxt, Price, Done]
[Choice]
Displ
[Coin10, Coin50, Coin100, CoinX]
CoinH(0,1): CoinHdlr
CoinIn
CoinCtl
CoinOut
Cash
Control(1,1)
Int
InpC
[Undo]
Pay
Plop
Pong
[Accept, Release, Change, Close]
[Money, Avail, Unavail, Flushed, Filled]
[Status, Complete, Maint]
Comm
CoinHdlr
[Coin10, Coin50, Coin100, CoinX]
Exists, Paid, CoinErr]
Flush
Sync

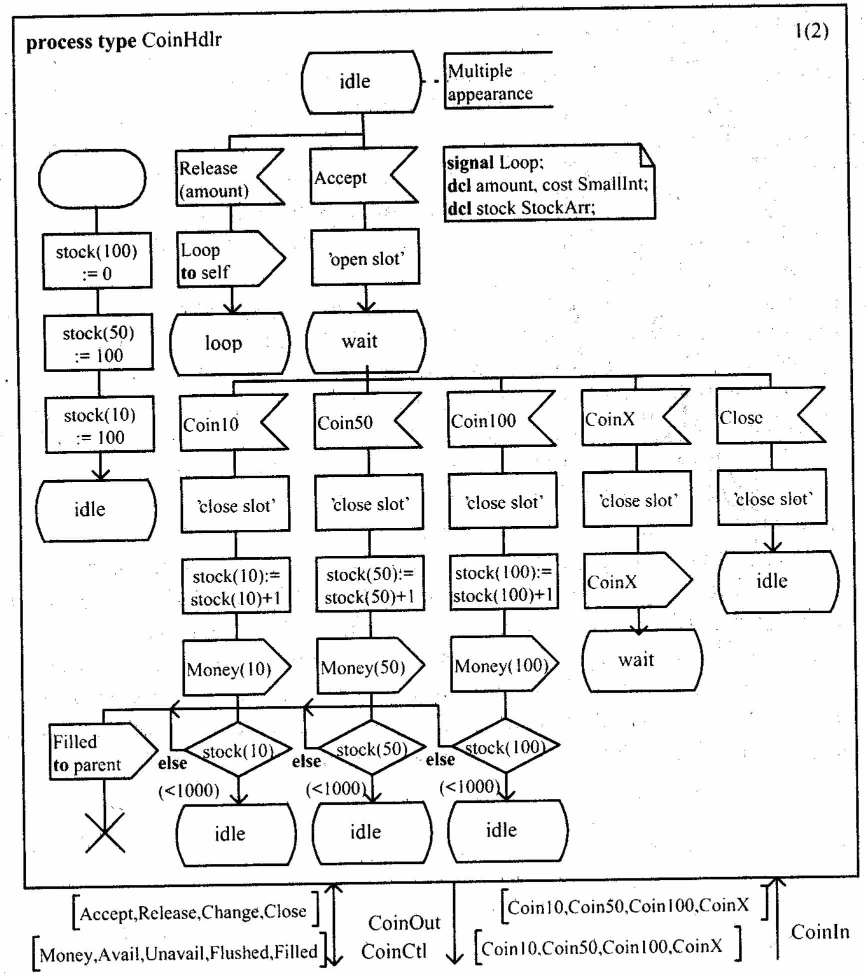
Process Control (1)



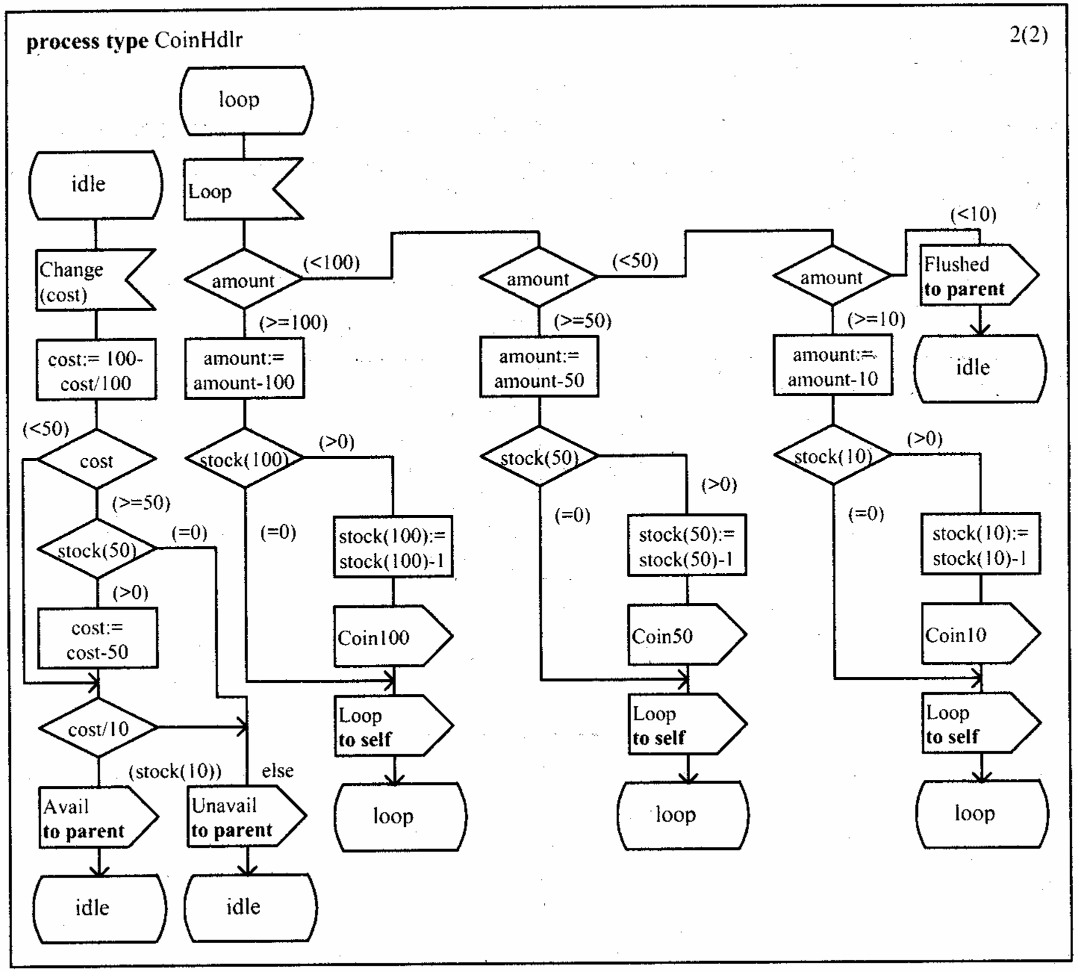
Process `Control` (2)



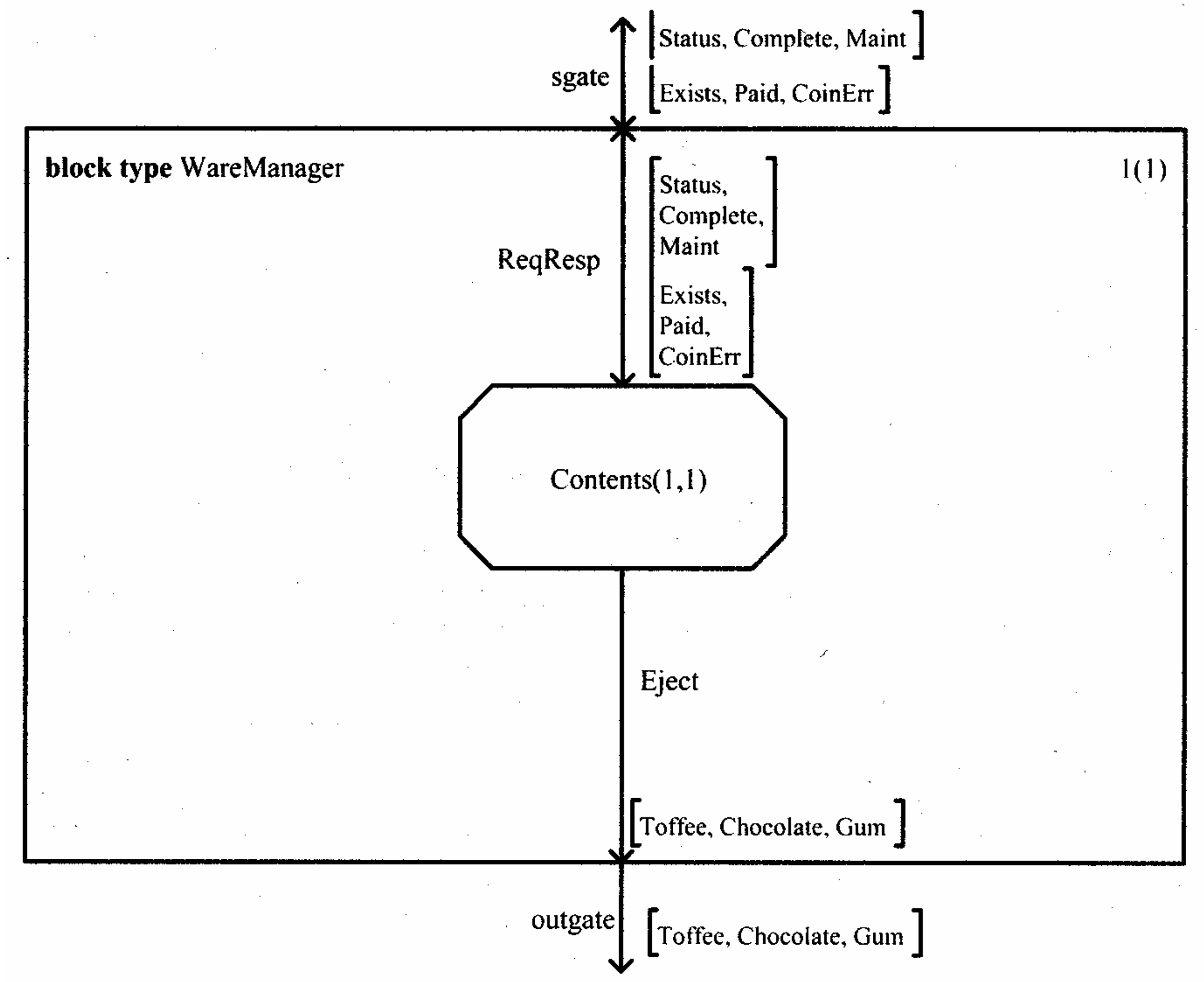
Process type `CoinHdlr` (1)



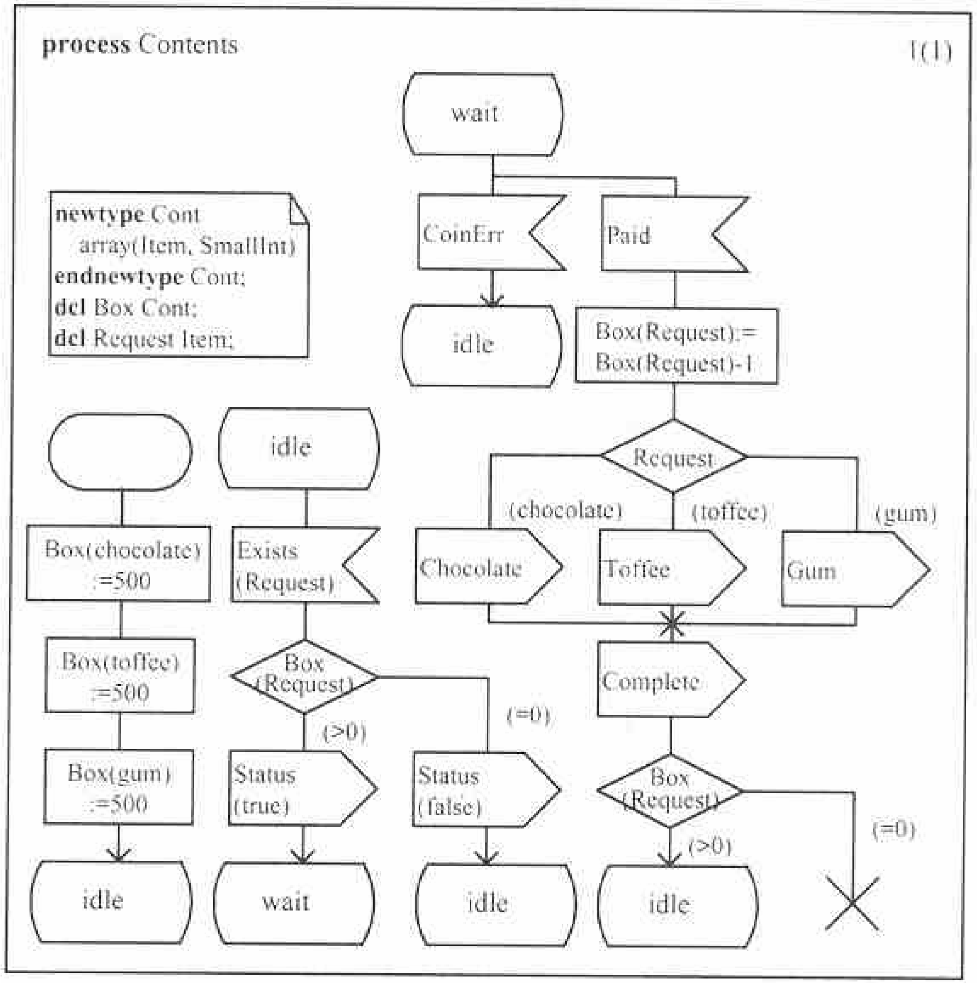
Process type CoinHdlr (2)



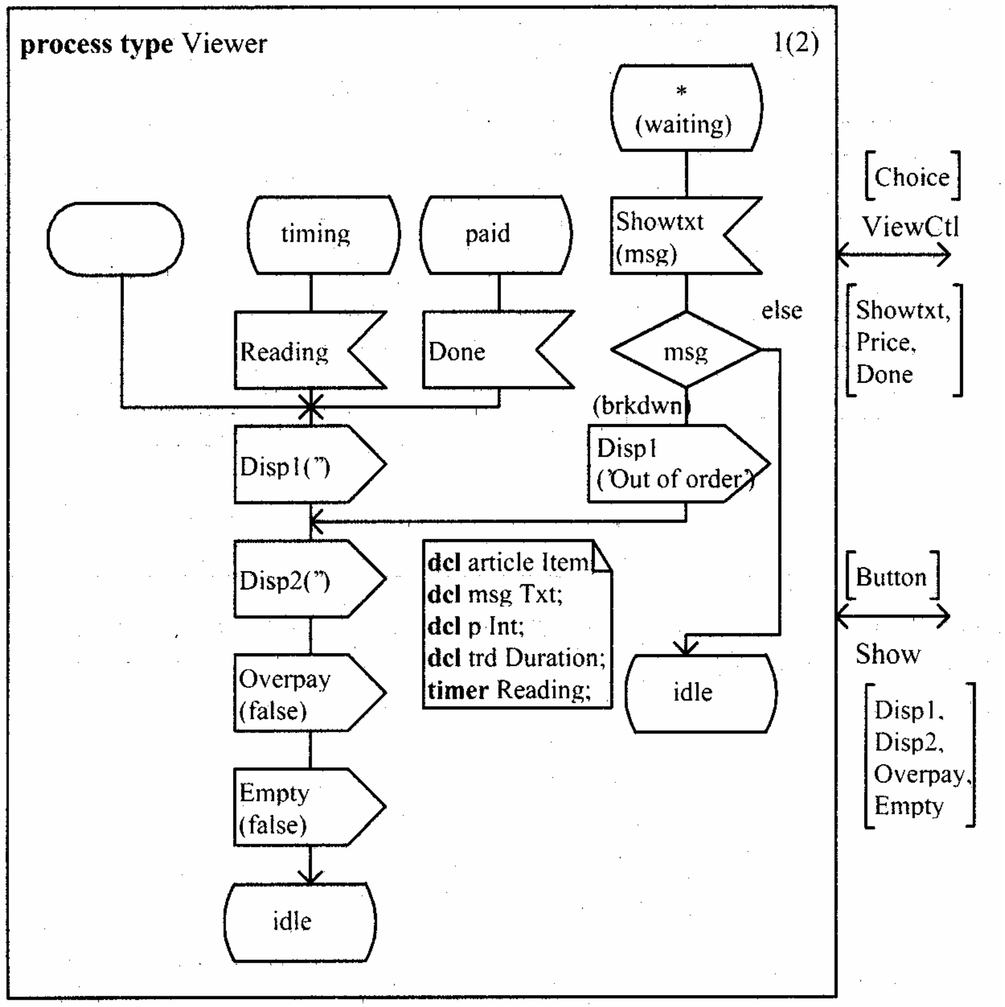
Block type WareManager



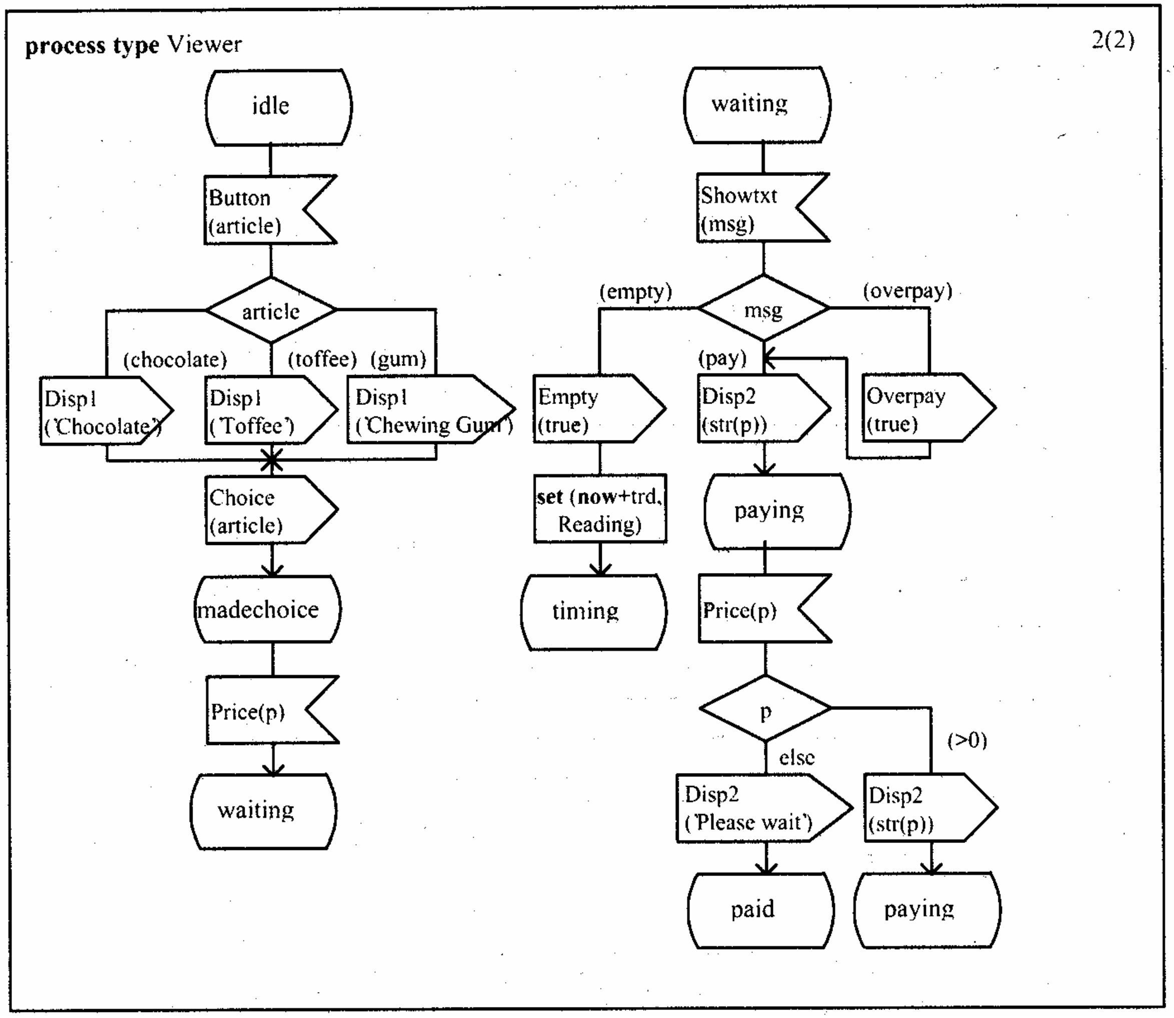
Process Contents



Process type `Viewer` (1)



Process type `Viewer` (2)



Comments on `ToffeeVendor` and `Dialogue`

`ToffeeVendor`

- Machine for selling toffee, chocolate, and (chewing) gum
- Accepts the coins `Coin10`, `Coin50` and `Coin100`
- `Dialogue` manages the dialogue with the user:
  - `InpC`: Selection by the user
  - `Pay`: Payment
  - `Flush`: Return of money
- `WMgr` manages the goods on stock:
  - `OutW`: Output of goods
  - `Sync`: Communication with `Dialogue`

`Dialogue`

- `CoinH` handles the coins:
  - `Plop`: Coins inserted by the user
  - `Pong`: Coins output by the machine
  - `Cash`: Communication with `Control`
- `ViewPt` manages user selections and outputs:
  - `Look`: Communication with the user
  - `Displ`: Communication with control
- `Control` is the control unit:
  - `Int`: Interrupt by the user
  - `Comm`: Communication with `WMgr`

Comments on `Control`

- Control of money and display
- Creation of the processes `CoinH` and `ViewPt`
- After user selection, initiate check whether requested good is present
- If present, display price and availability of change
- Accept coins until price is payed
- Return change (if any)
- In case of time out or interrupt, terminate current action and return money
- In case of maintenance or full stock of coins, lock coin slot and terminate process

Comments on `CoinHdlr` and `Contents`

`CoinHdlr`

- Initialization of coin stock
- On reception of `Accept` signal, receive coins and update stock until a wrong coin is inserted or the `Close` signal is received
- Signal a full stock of coins to `Control`
- On reception of `Release`, return the specified amount
- On reception of the `Change` signal, check whether sufficient change is available

`Contents`

- Initialization of ware storage
- On reception of the `Exists` signal, reply whether the requested good is present
- On reception of the `Paid` signal, output requested good and update ware storage
- Terminate if the ware storage is exhausted

Advanced Concepts

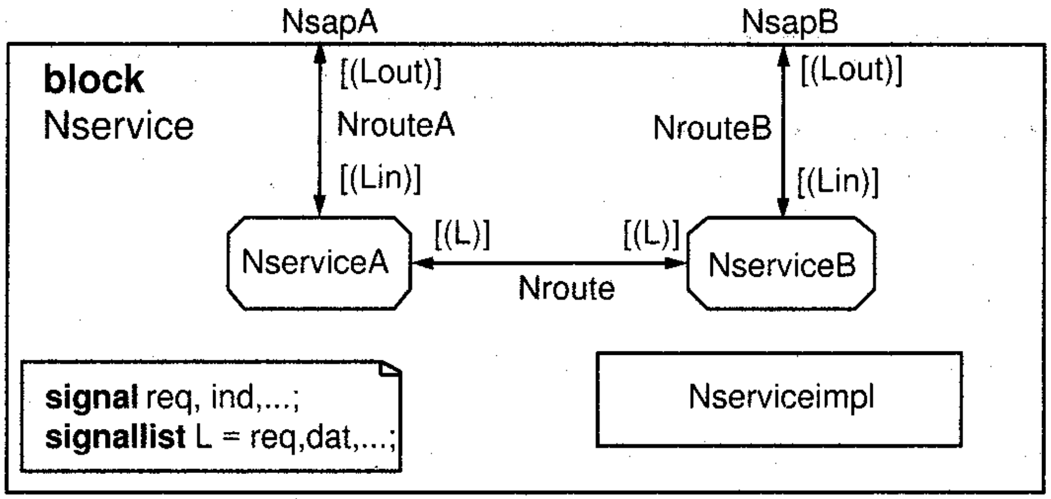
Combined block specifications

- Up to now: Refinement of a block *either* by a block interaction diagram *or* a process interaction diagram
- **Combined block specification**: Refinement *both* by a block interaction diagram *and* a process interaction diagram
- Refinements are alternatives and should be equivalent (which cannot be guaranteed)
- Interpretation of the specification on a **consistent cut**:
  - In case of selecting a process interaction diagram, refinements ends
  - Otherwise, the refining blocks are taken into account (interpretation on a more detailed level)

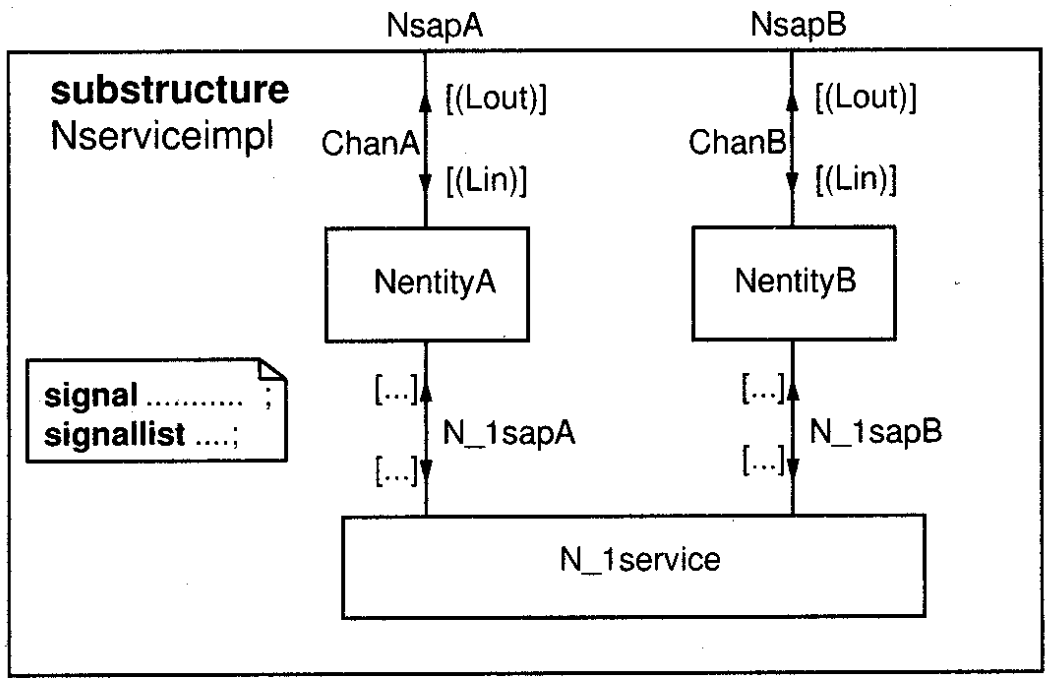
Example: Protocol specification

- Protocols in telecommunication are structured into (abstraction) **layers**
- On each layer, communication is specified by a process interaction diagram
- Communication is realized on the next lower layer $\Rightarrow$
  Specification by a block interaction diagram

Process interaction diagram



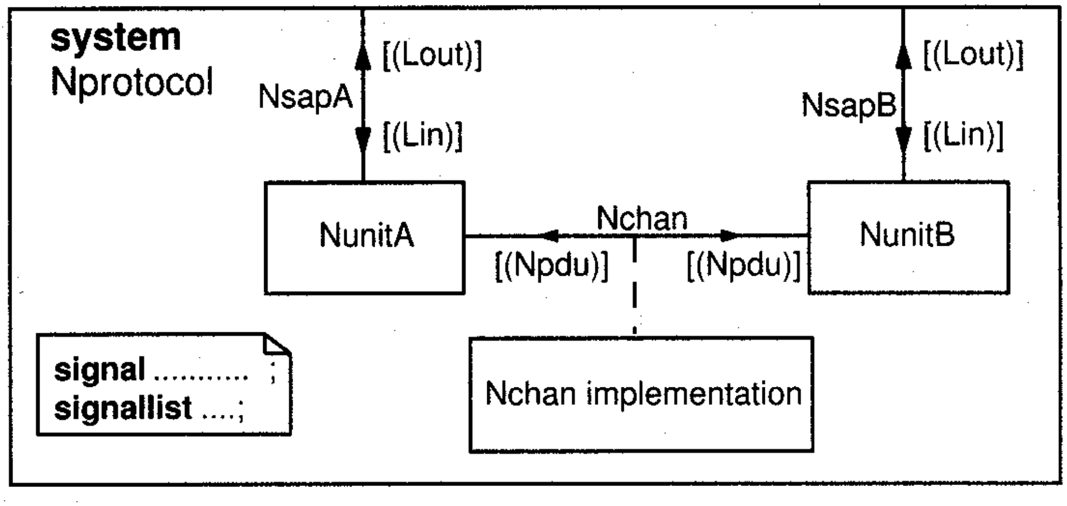
Block interaction diagram



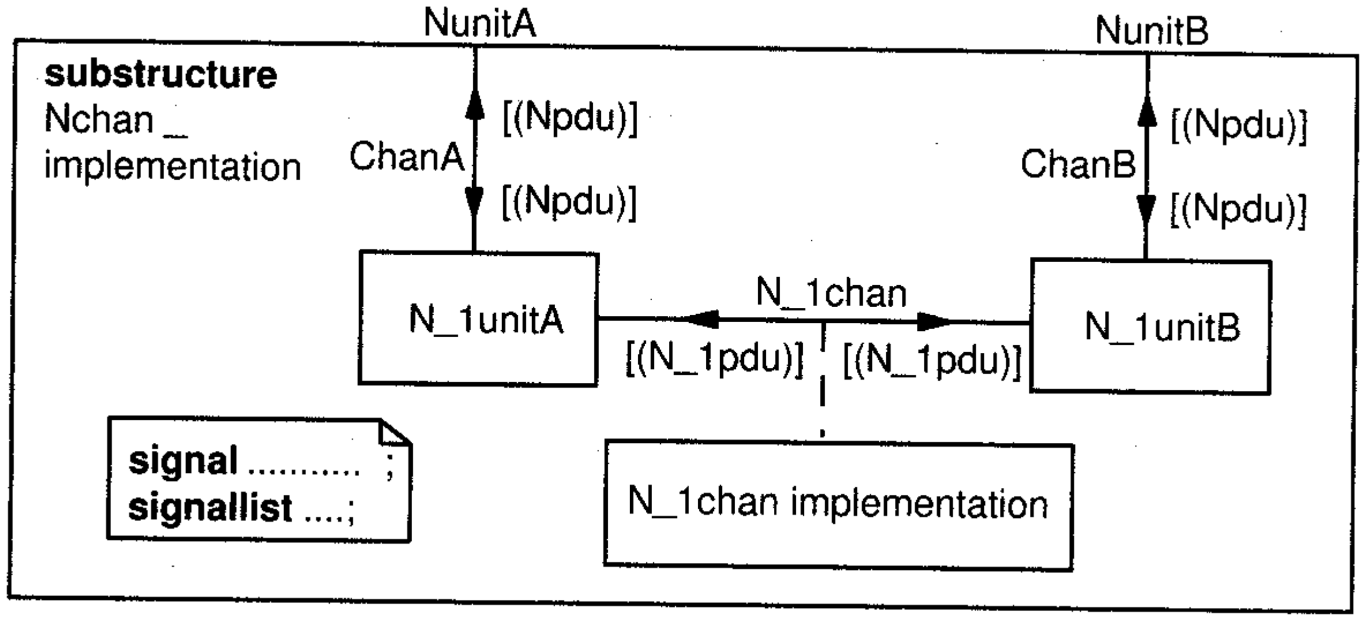
Combined channel specification

- Channels may be refined like blocks
- A channel is refined by an interaction diagram
- The blocks to be connected serve as interfaces of the interaction diagram

Example of a combined channel specification



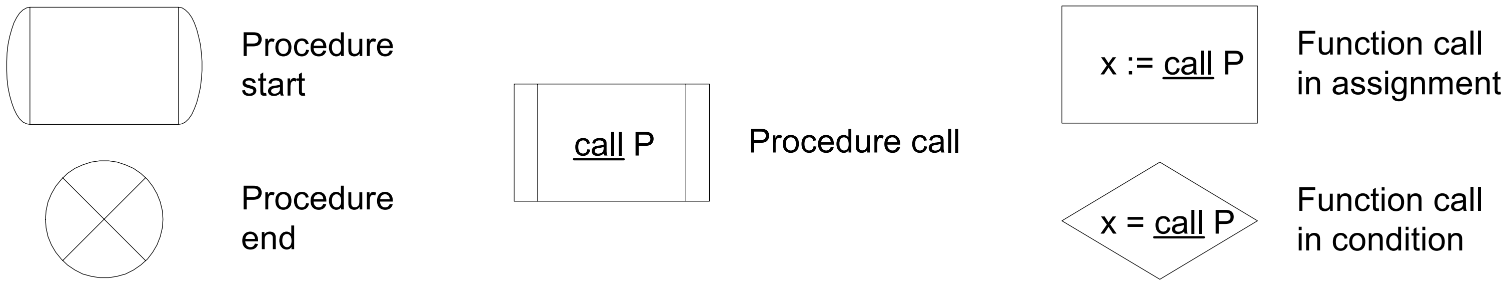
Refinement of a channel



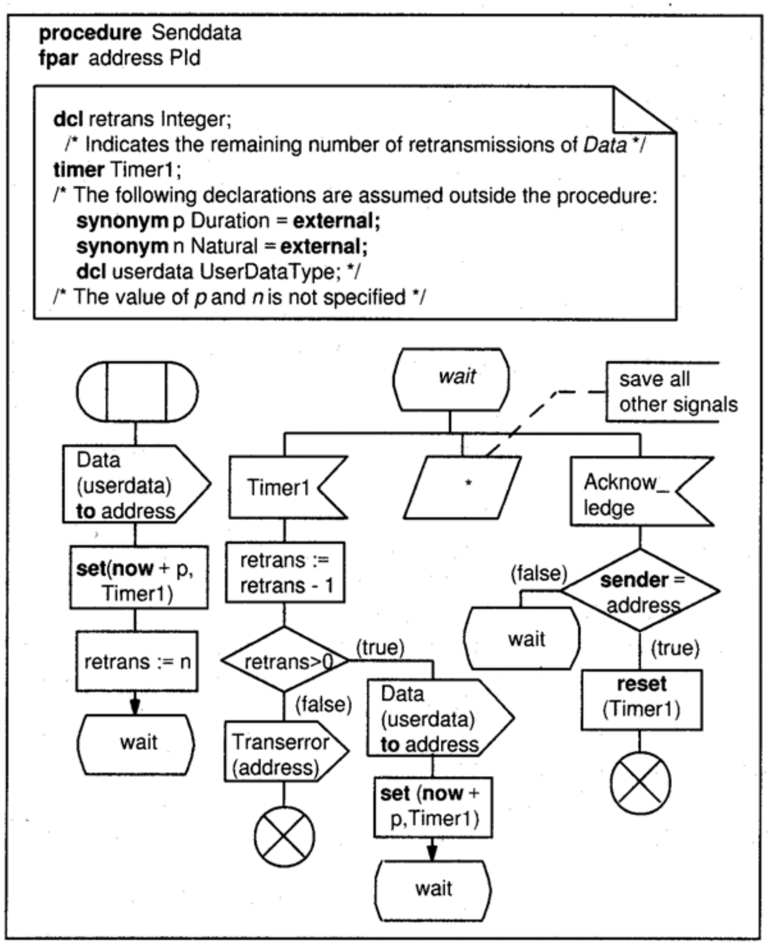
Procedures

- A **procedure** serves to structure local computations in a process
- A procedure may have formal **parameters** which are replaced with actual parameters when the procedure is called
- Similarly to a process, a procedure is defined by an extended state machine
- When a procedure is called, the calling process is suspended until the procedure call has been executed
- During execution of the call, the input queue and the variables of the calling process may be executed
- In addition to local calls, there are also **remote procedure calls**

Graphical notation of procedures



Example of a procedure specification

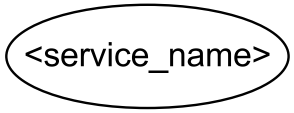


procedure Senddata
fpar address PId

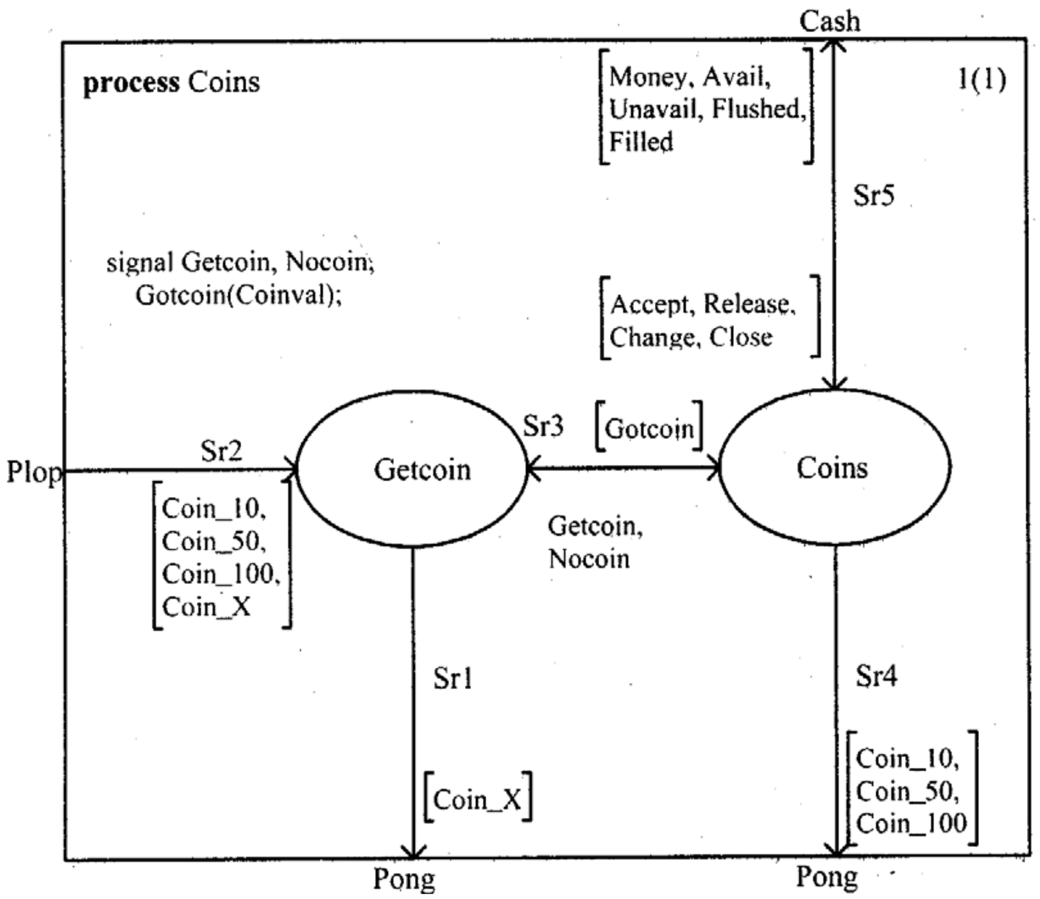
dcl retrans Integer;
/* Indicates the remaining number of retransmissions of *Data* */
timer Timer1;
/* The following declarations are assumed outside the procedure:
synonym p Duration = **external;**
synonym n Natural = **external;**
dcl userdata UserDataType; */
/* The value of *p* and *n* is not specified */

Services

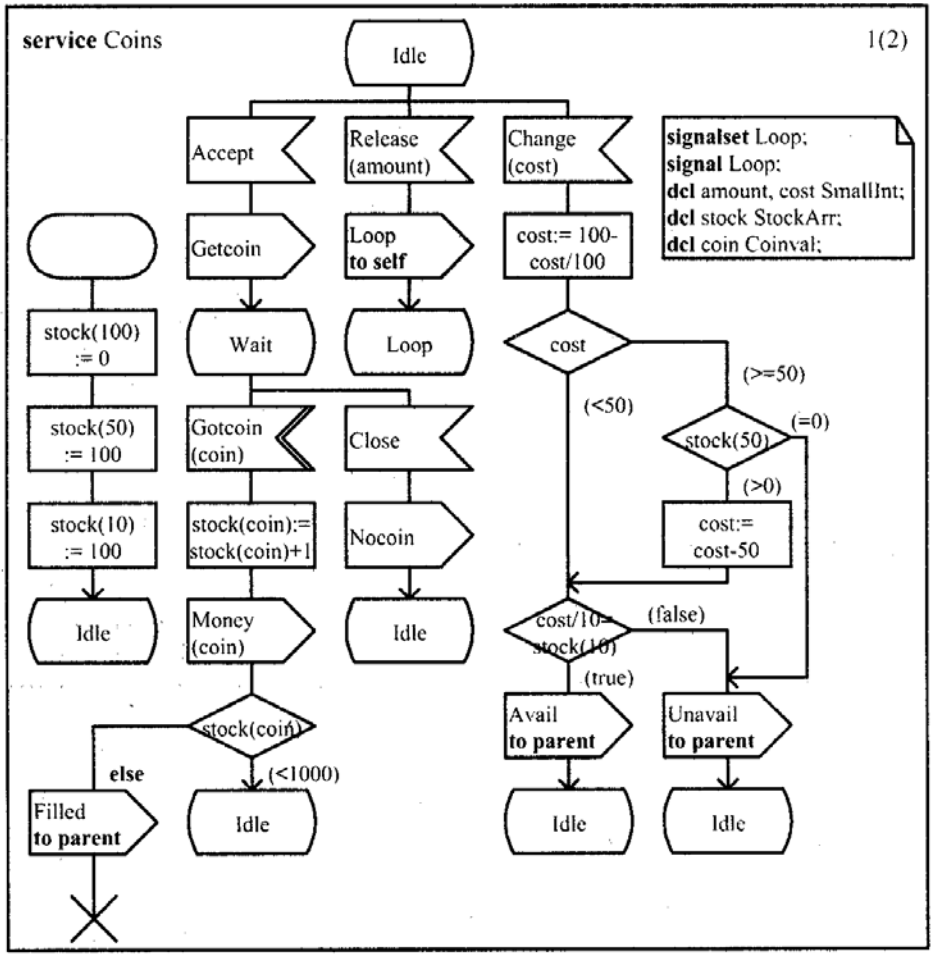
- Instead of an extended state machine, a process may be described by a **service interaction diagram**
- In this case, a process consists of a set of interacting **services**
- Concurrency within a process is prohibited, i.e., at each point in time at most one service may perform a state transition
- Services share the variables and the input queues of the process
- For each signal, there may be only one service which handles this signal
- Graphical notation of a service reference:



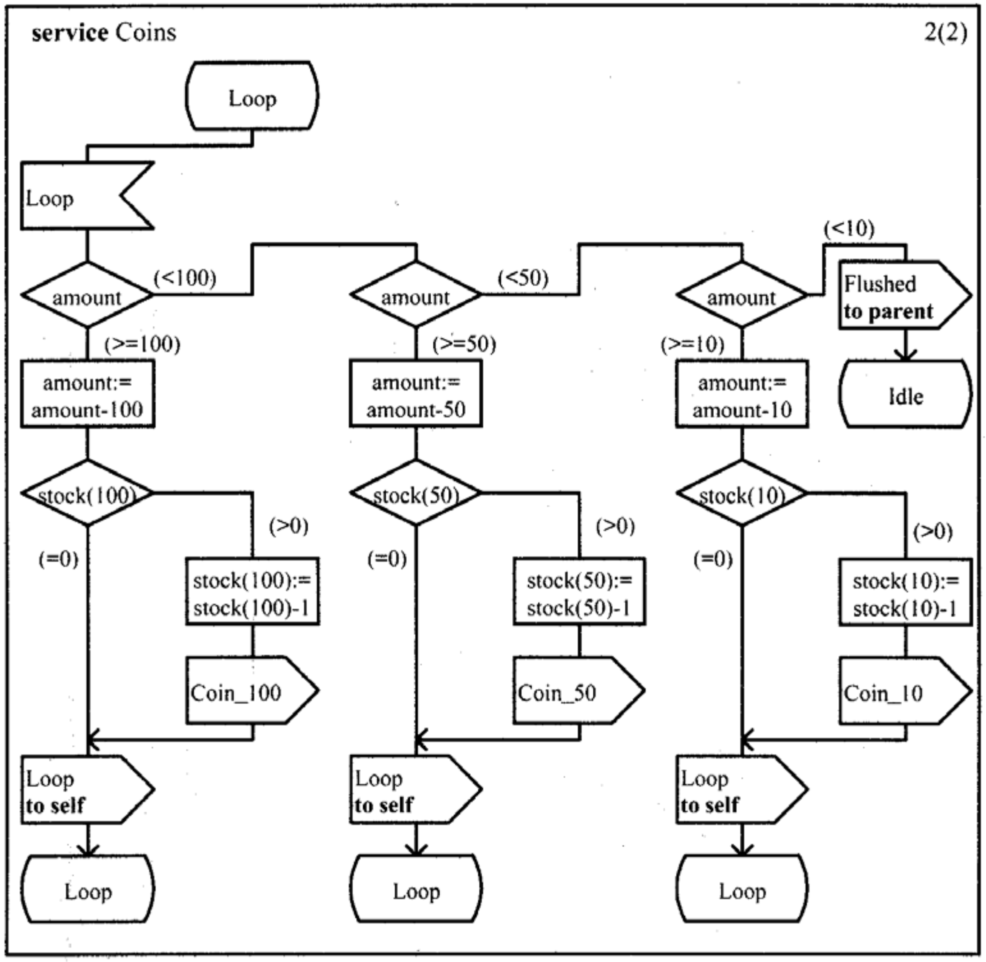
Service interaction diagram for the process `Coins`


Cash
process Coins
1(1)
Money, Avail, Unavail, Flushed, Filled
Sr5
signal Getcoin, Nocoin; Gotcoin(Coinval);
Accept, Release, Change, Close
Sr3
Gotcoin
Plop
Sr2
Getcoin
Coins
Coin_10, Coin_50, Coin_100, Coin_X
Getcoin, Nocoin
Sr1
Sr4
Coin_X
Coin_10, Coin_50, Coin_100
Pong
Pong

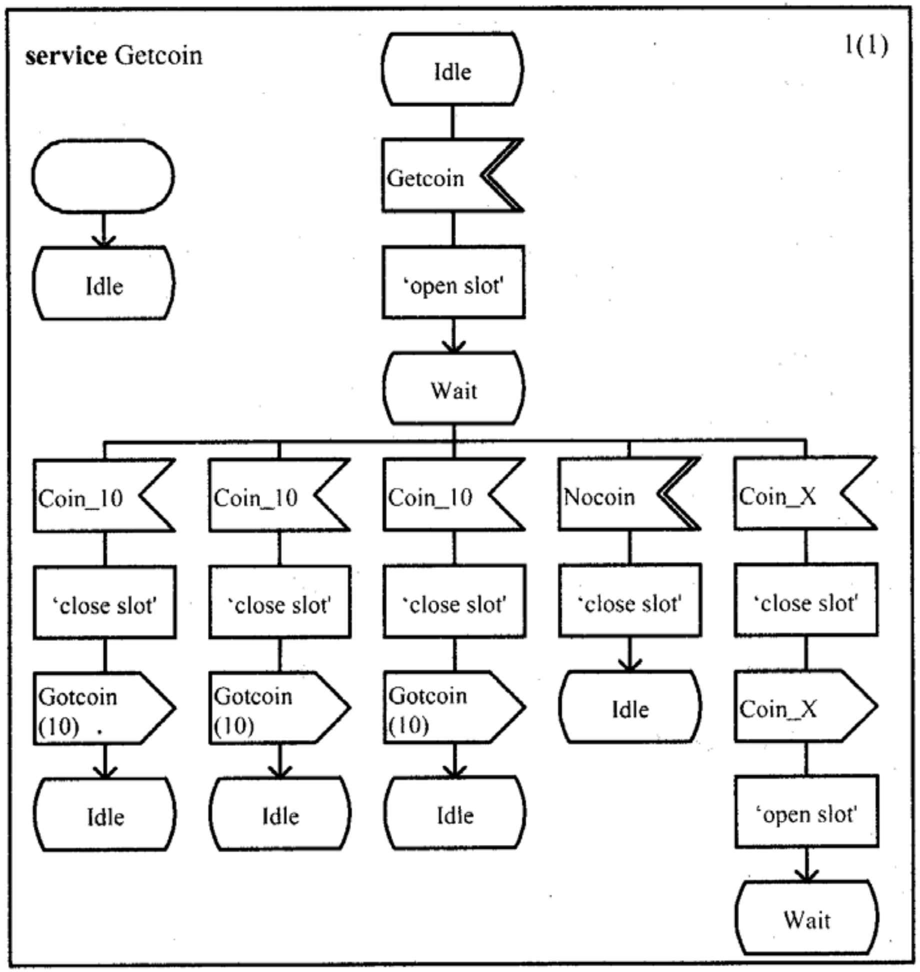
Service `Coins` (1)



Service `Coins` (2)



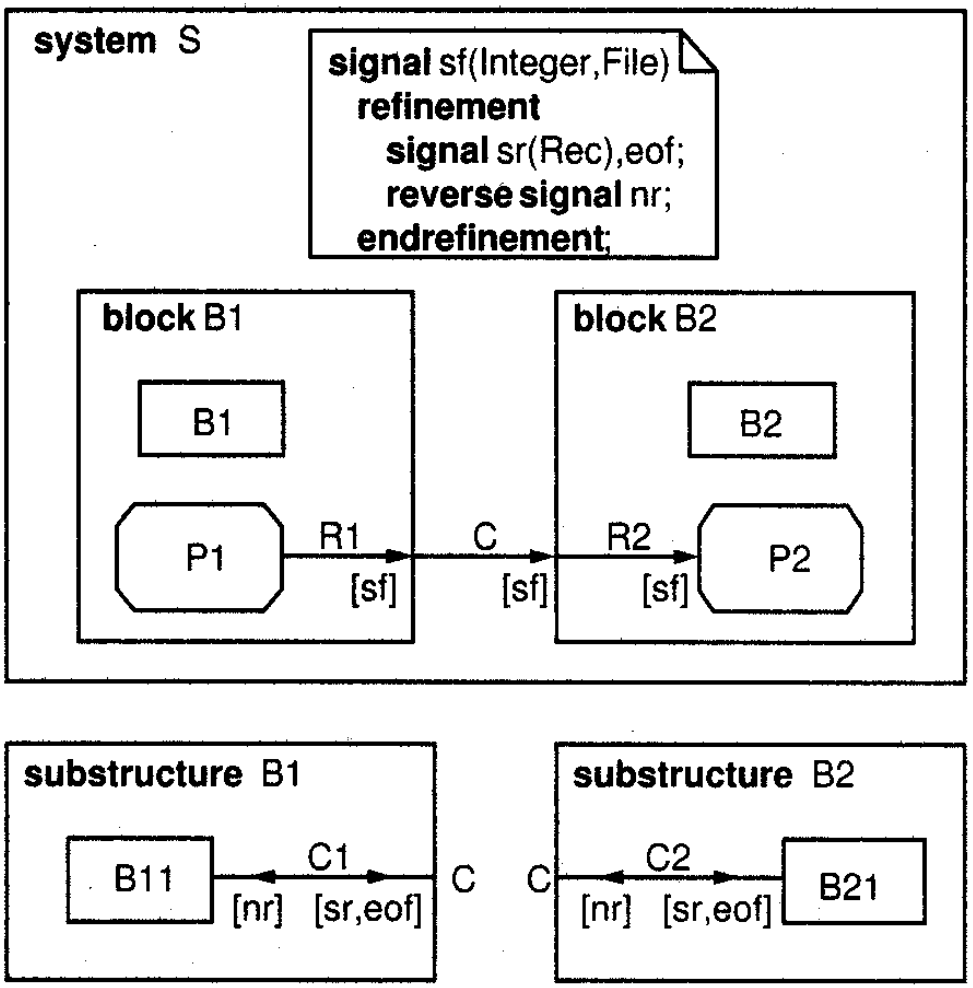
Service GetCoin



Refinement of signals

- A signal may be **refined** by a set of signals
- A channel transmitting a refined signal automatically transmits all of its refinements
- In the refinement, bi-directional signals are permitted even for uni-directional channels
- Refinement does not have a dynamic semantics:
  - Signals are not composed or decomposed dynamically
  - At runtime, *either* the complex signal *or* the refining signals may be transmitted
  - Therefore, it is not allowed to refine only one end of a channel when defining a consistent cut for execution

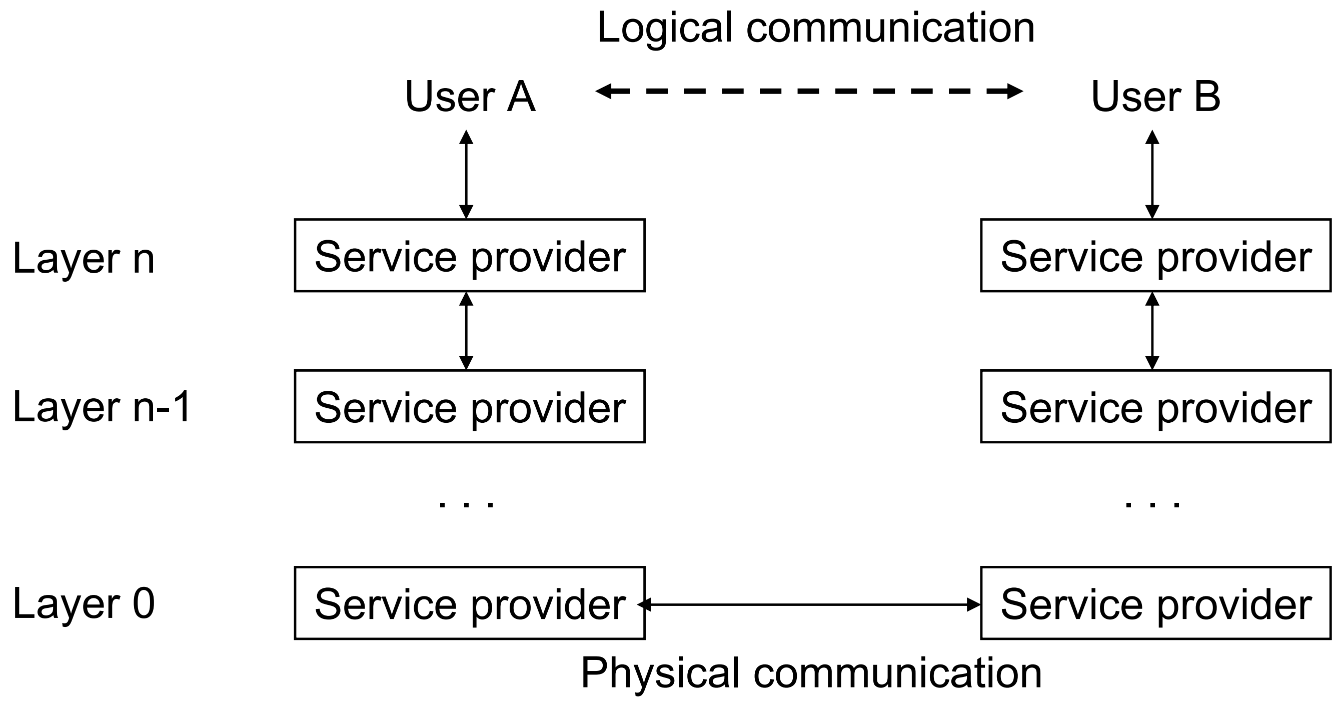
Example of the refinement of signals



Specification of Protocols

Layered architecture of telecommunication systems

- Eventually, data must be transmitted on physical channels
- However, the user would like to send **logical objects** (e.g., files)
- Solution: layered architecture with multiple levels of abstraction



Example: OSI reference model

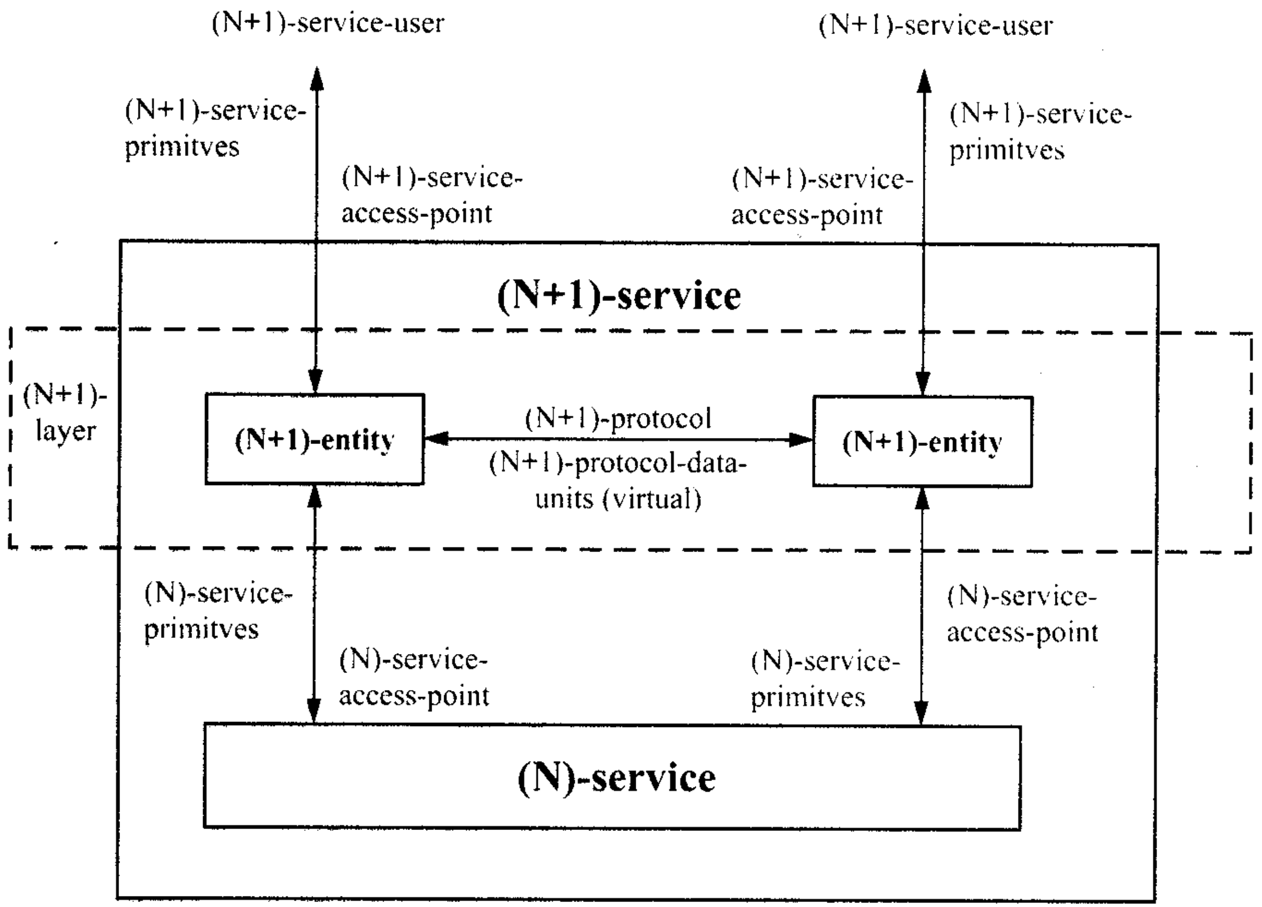
OSI: Open Systems Interconnection Standard (ISO)

Application layer	(e.g., File transfer)
Presentation layer	Data compression
Session layer	Session management
Transport layer	Ordering, flow control
Network layer	Routing
Data link layer	Error correction
Physical layer	Transmission of bit streams

Basic notions

- **(N)-service**: Service on layer N
- **(N)-service provider**: Provider of the service
- **(N)-service user**: User of the service
- **(N)-service access point**:
  Access point which provides the service provider to the service user
- **(N)-service primitive**: Communication primitive of a service
- **(N)-service data unit**: Transport unit for data on layer N
- **(N)-layer**: Layer N
- **(N)-protocol**: Providing for providing the service
- **(N)-entity**: Entity on layer N
- **peer entity**: Entity on the same level (“partner”)
- **(N)-protocol data unit**: Data unit of the protocol on layer N

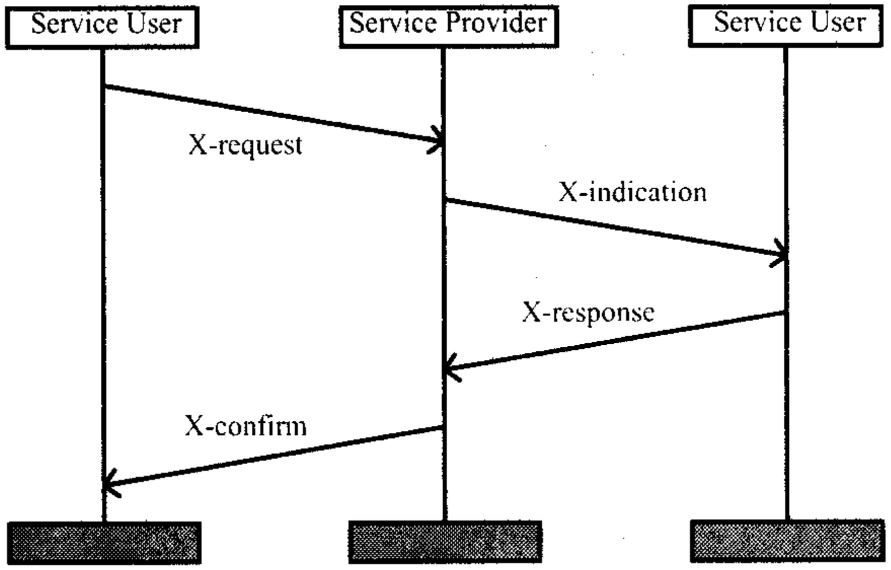
Generic model of layers



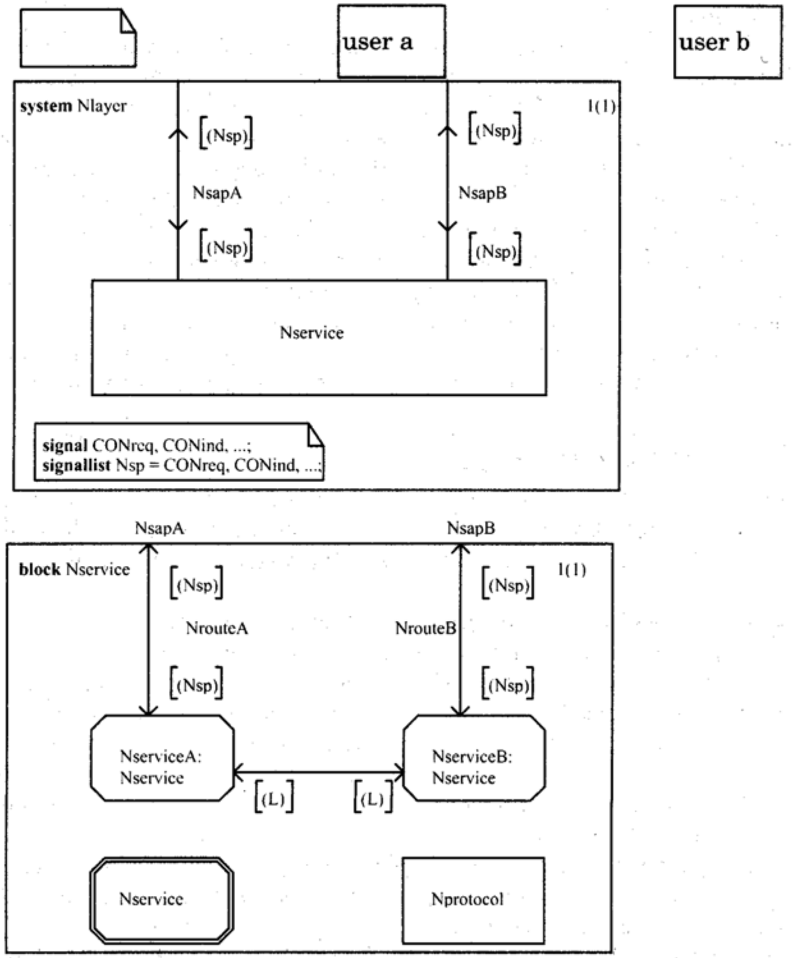
Service primitives

- **request**: Service user requests an action
- **indication**: Service provides indicates the request to the partner
- **response**: Partner responds to the indicated request
- **confirm**: Service confirms that the request has been delivered

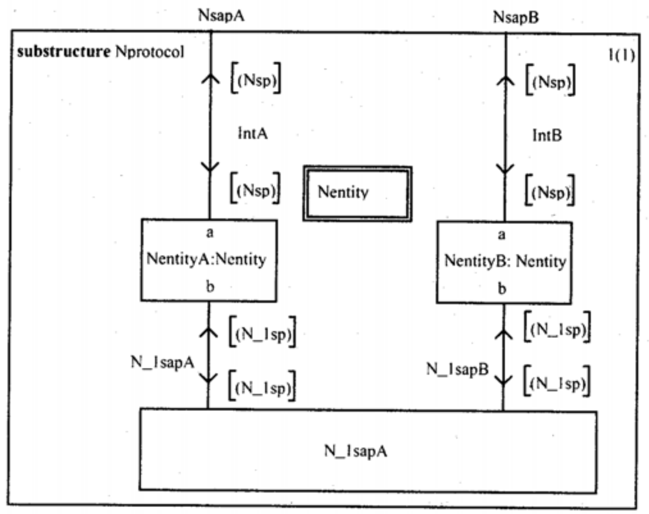
Message Sequence Chart (MSC)

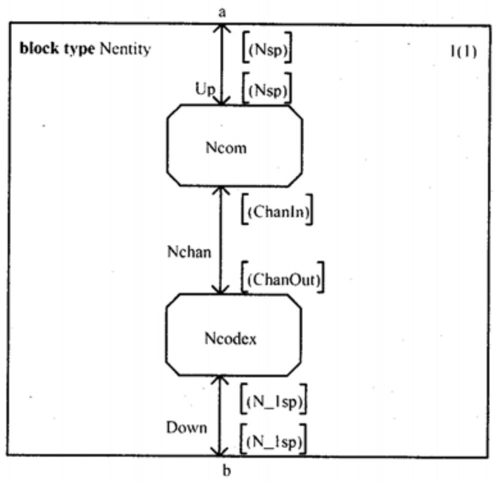


Service specification in SDL

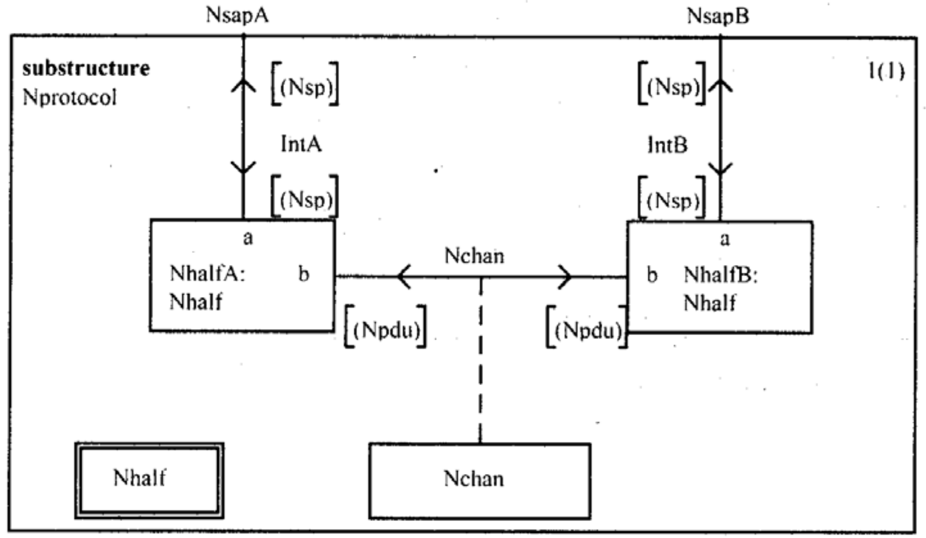


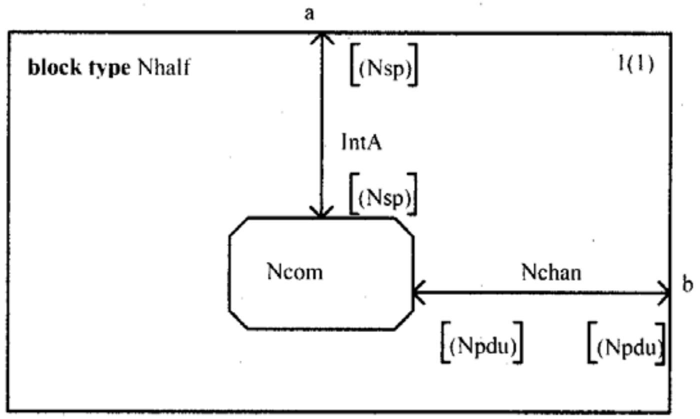
Protocol specification in SDL



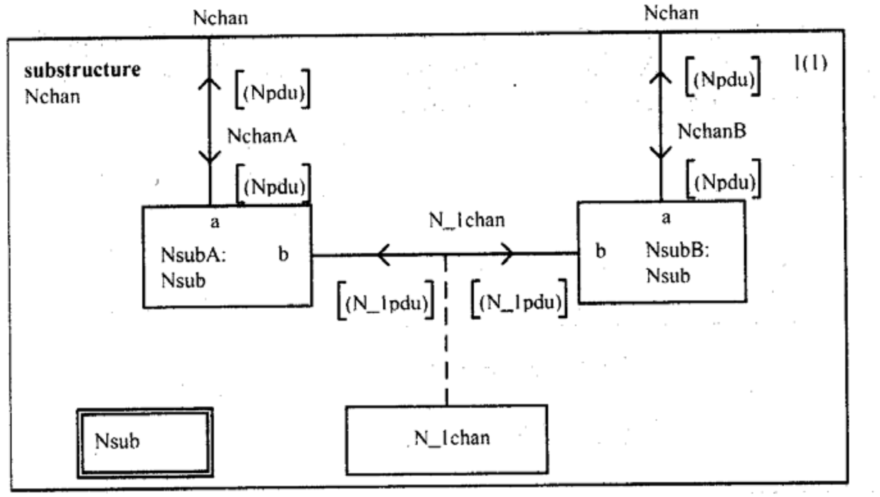


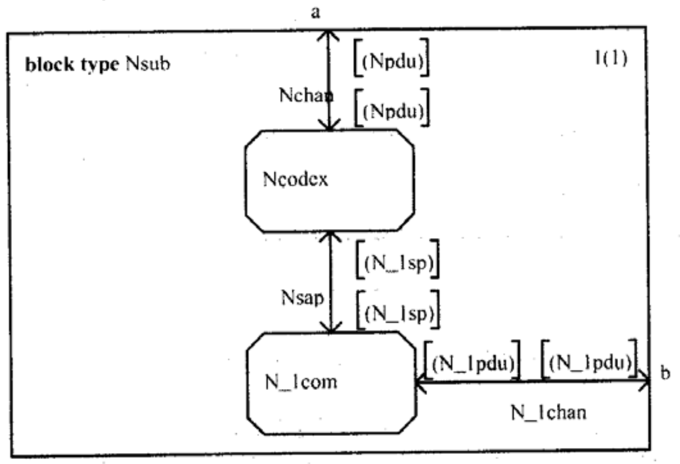
Alternative specification with a complex channel (1)





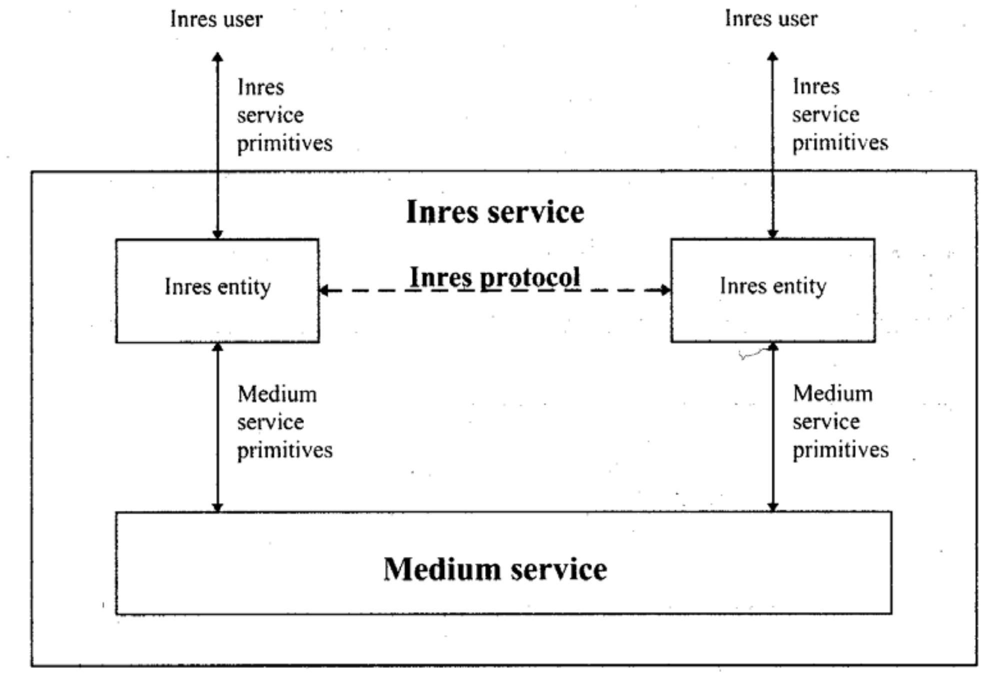
Alternative specification with a complex channel (2)





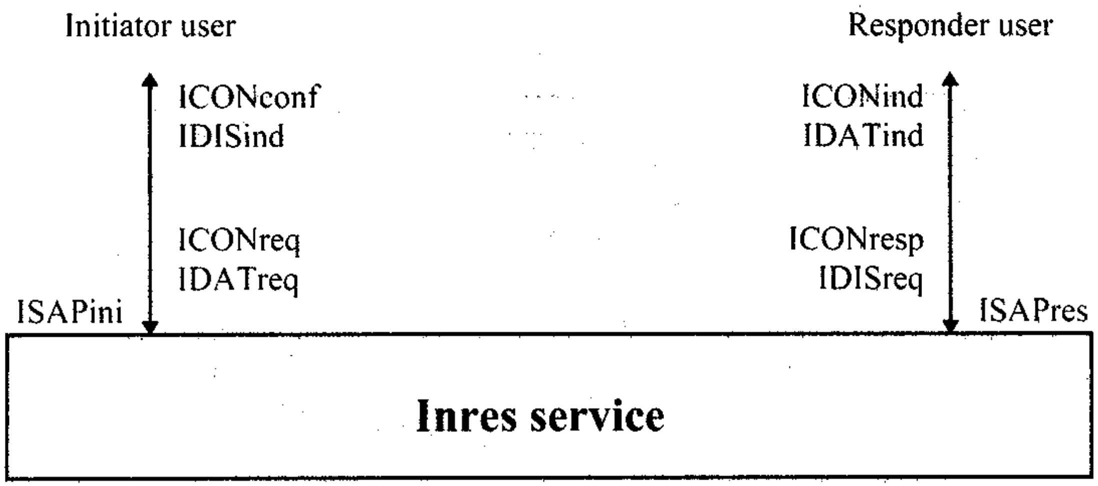
Example: Inres system

- **Inres**: Initiator - Responder
- **Connection-oriented service** for transmitting data
- Use of a **connection-less medium** (potential data losses)



Survey of service primitives

- **ICONreq**: Initiator requests connection
- **ICONind**: Service provider indicates this to the responder
- **ICONres**: Responder replies
- **ICONconf**: Service provider confirms the connection
- **IDATreq(ISDU)** : Initiator sends data
- **IDATind(ISDU)** : Service provider transmits data to the responder
- **IDISreq**: Responder aborts connection
- **IDISind**: Service provider indicates this to the initiator



Behavior of the Inres service (1)

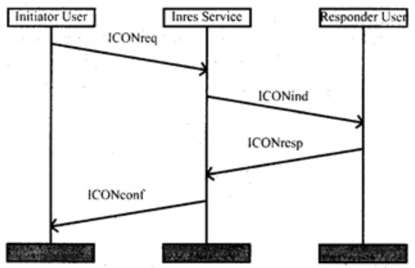


Figure 10.9a *Successful connection establishment*

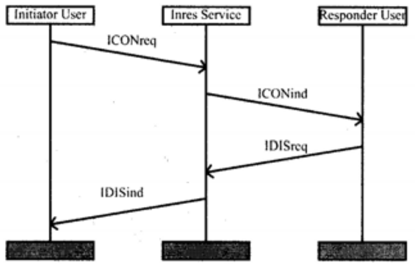


Figure 10.9b *Unsuccessful connection establishment (rejection by the Responder)*

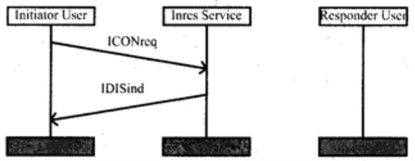


Figure 10.9c *Unsuccessful connection establishment (erroneous transmission of the connection request)*

Behavior of the Inres service (2)

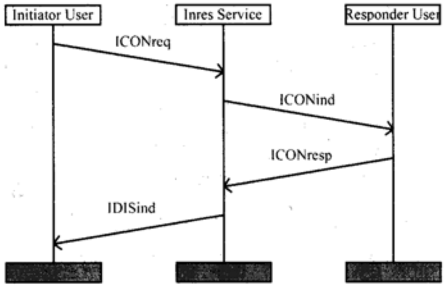


Figure 10.9d *Unsuccessful connection establishment (erroneous transmission of the connection response)*

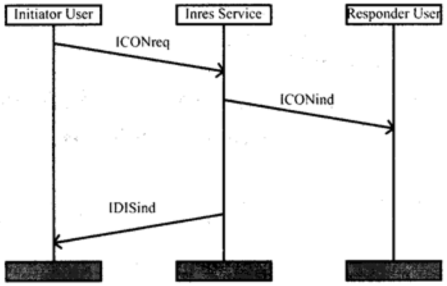


Figure 10.9e *Unsuccessful connection establishment (Responder ignores connection request)*

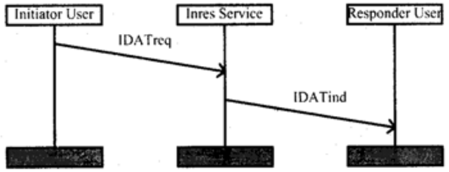


Figure 10.9f *Successful data transfer*

Behavior of the Inres service (3)

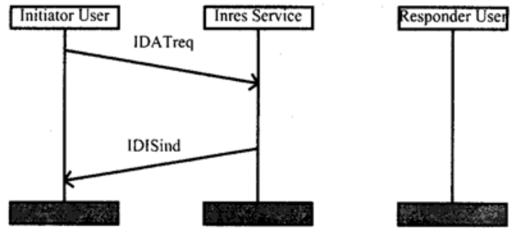


Figure 10.9g *Unsuccessful data transfer (erroneous transmission of data)*

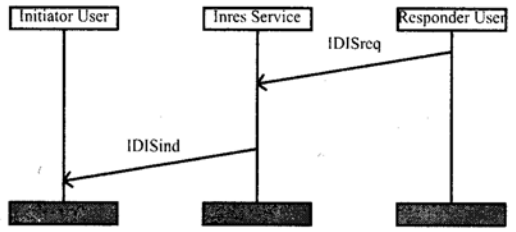


Figure 10.9h *Successful disconnection*

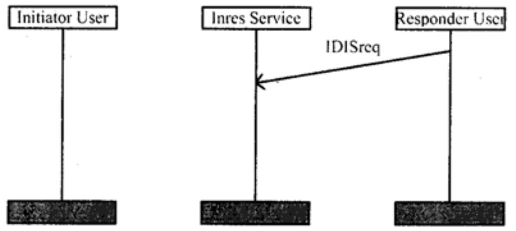


Figure 10.9i *Unsuccessful disconnection 'erroneous transmission of disconnection request)*

Behavior of the medium

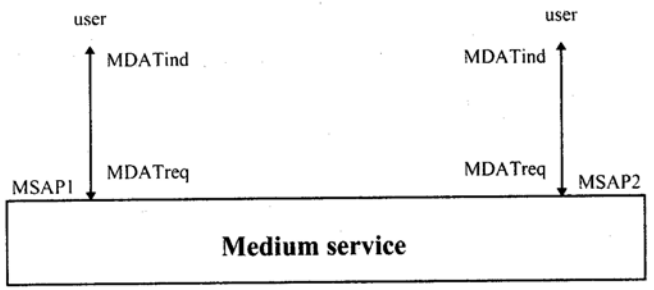


Figure 10.10 *The Medium service*

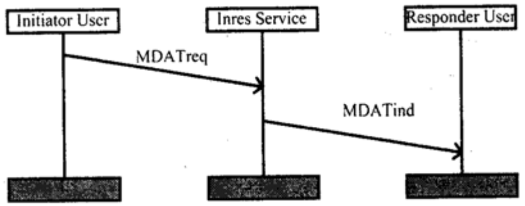


Figure 10.11a *Successful data transfer*

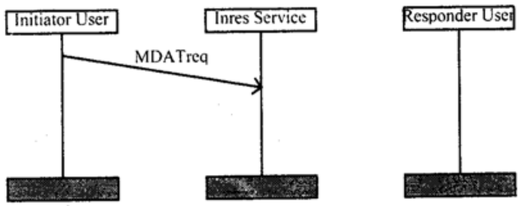
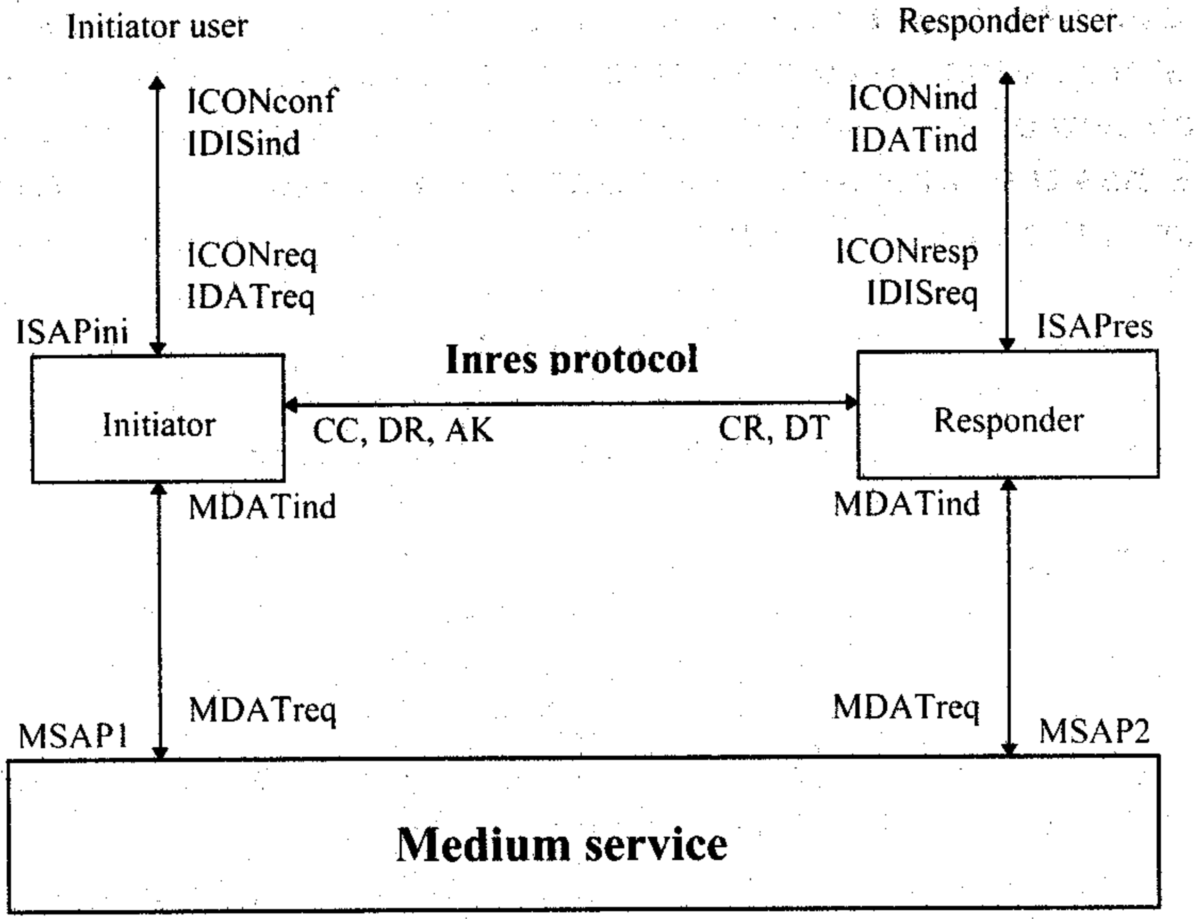


Figure 10.11b *Unsuccessful data transfer (erroneous transmission of data)*

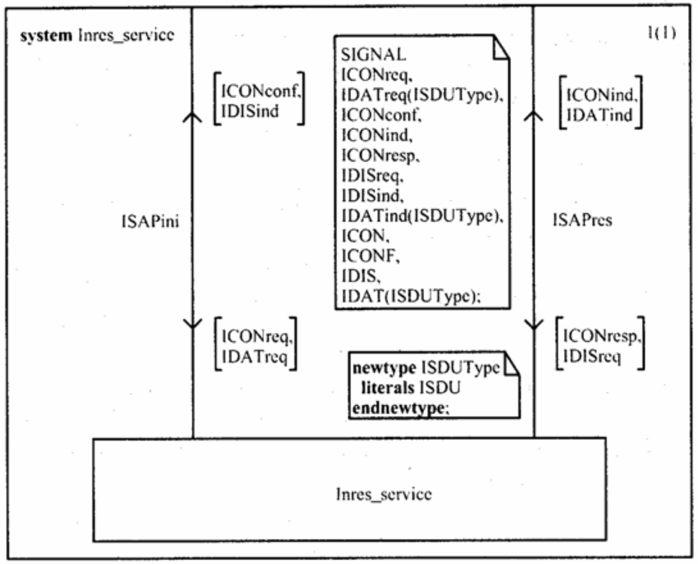
The Inres protocol

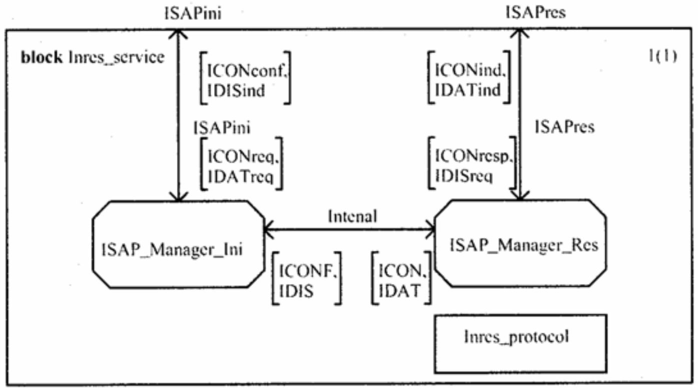


Operation of the protocol

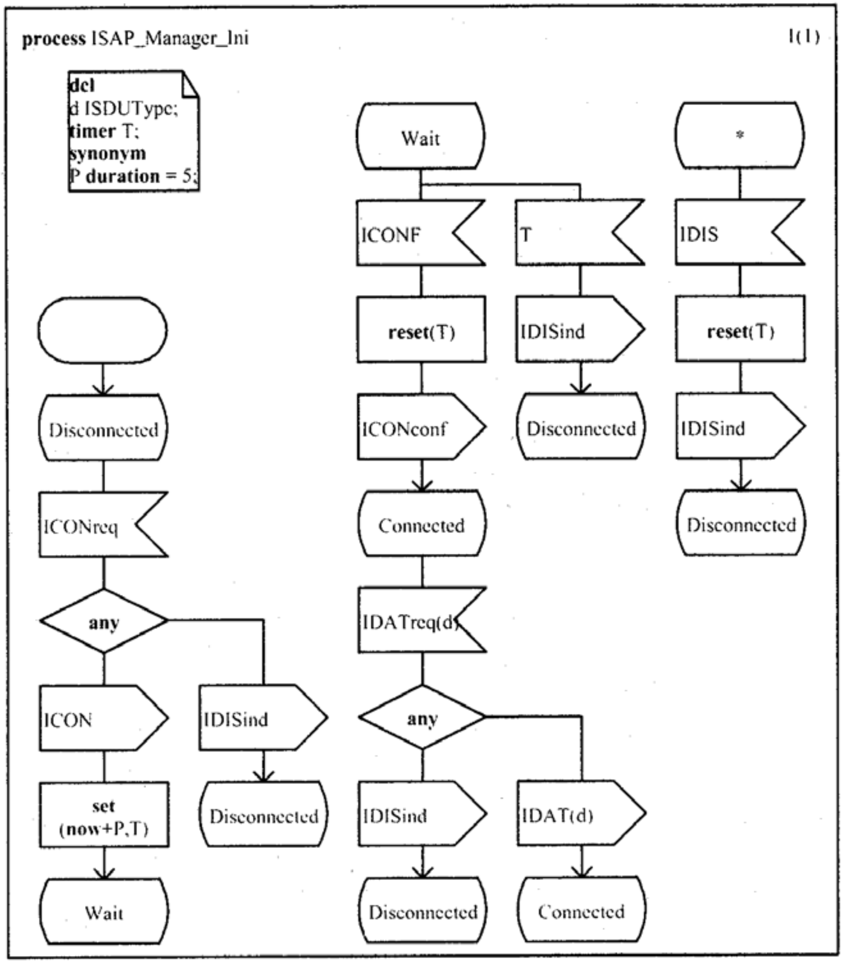
- **Establish connection**:
  After initiation through ICONreq the connection is established, if possible. In case of errors (e.g., timeouts), the attempt to establish a connection is aborted.
- **Data transfer**:
  After initiation by IDATreq data are transferred, if possible. Each receipt is acknowledged. In case of timeouts, the transfer of data is attempted multiple times. In order to recognize duplicate transfers, data are marked with an alternating bit.
- **Release connection**:
  The connection may be released by the responder or by the service provider.

Inres system

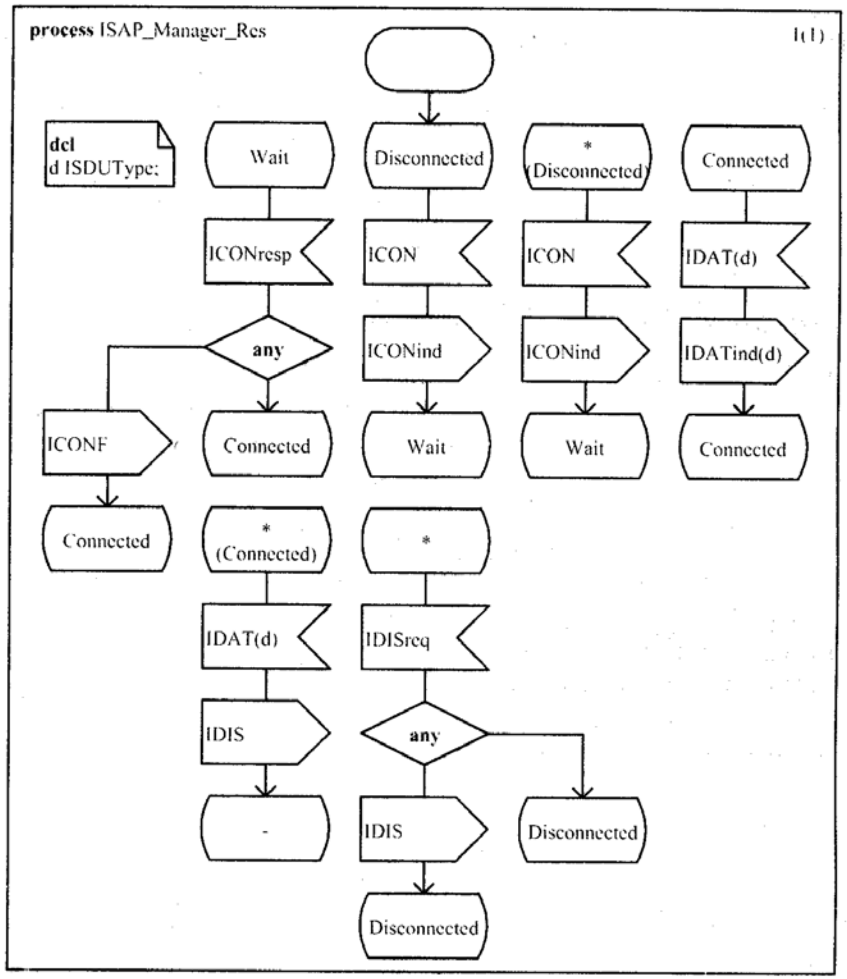




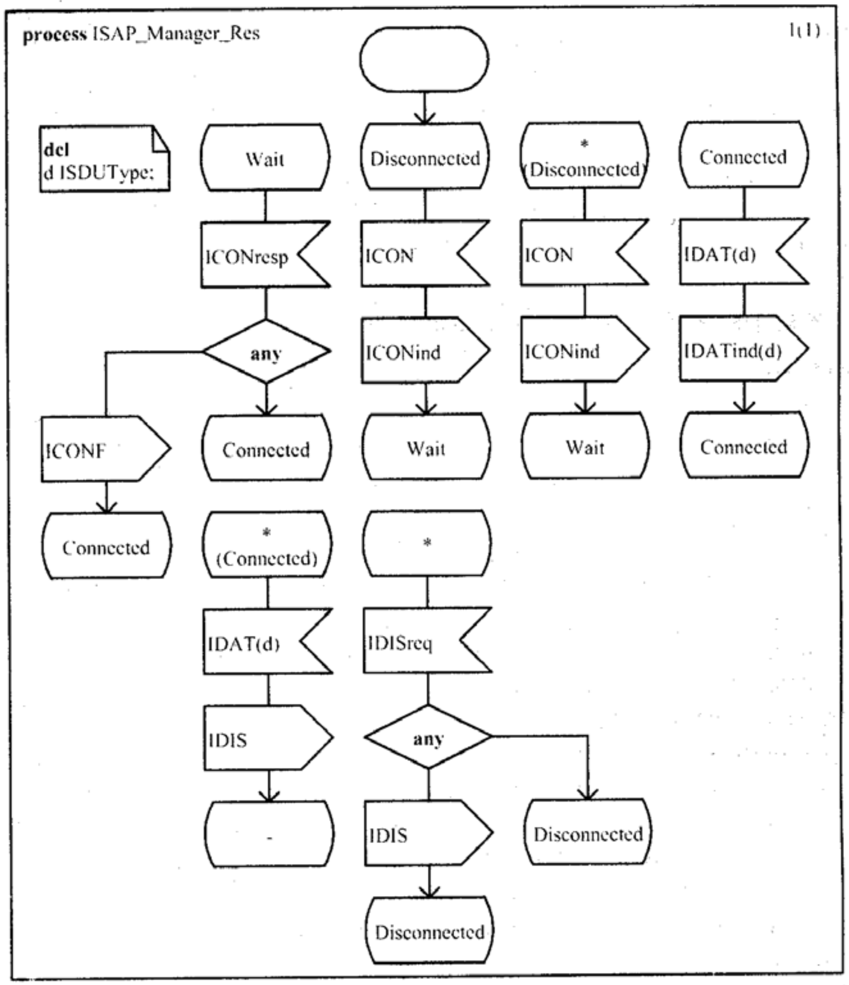
Process ISAP_Manager_Ini



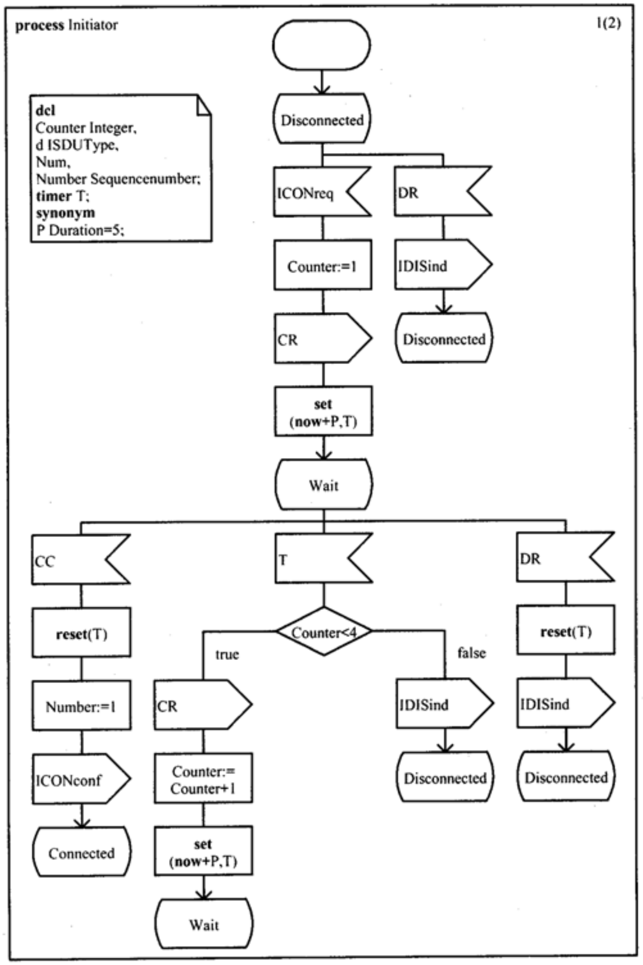
Process ISAP_Manager_Res



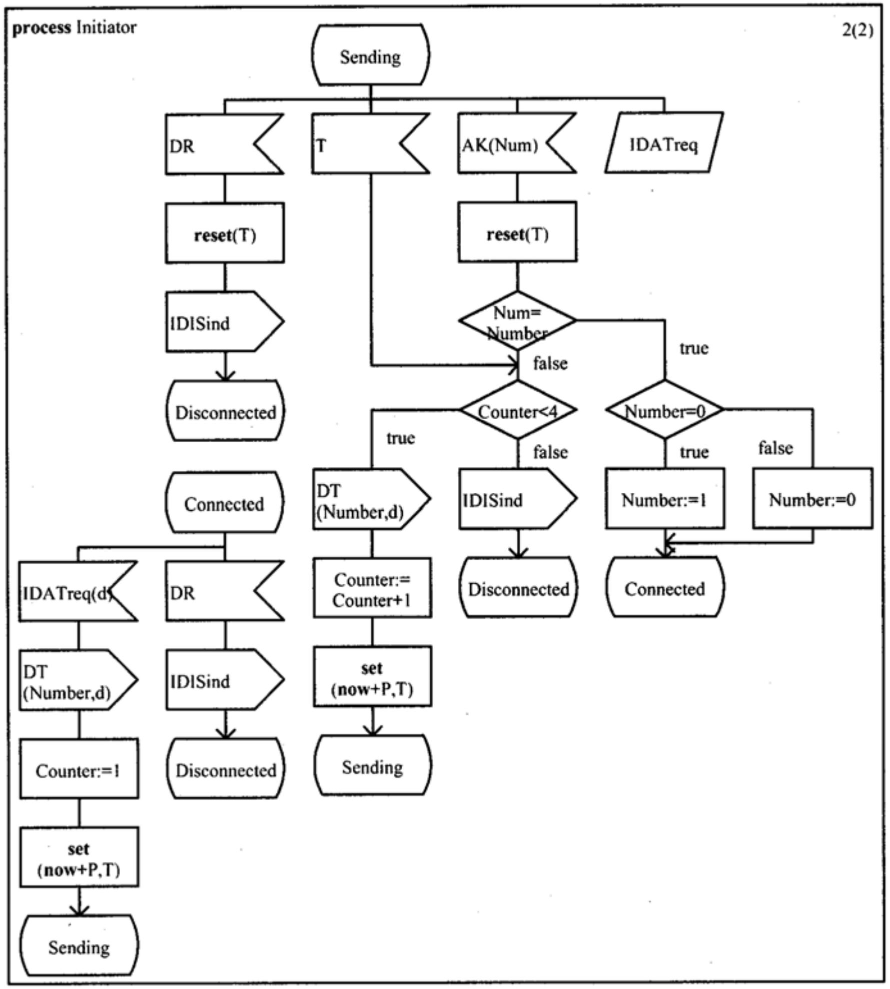
Inres protocol



Process Initiator (1)



Process Initiator (2)



Summary

- SDL is a standard for the specification of telecommunication systems
- There are industrial applications of SDL
- SDL has a long history
- In particular, it is well suited for layered communication architectures
- SDL specifications are operational $\Rightarrow$
  Generation of code from the specification
- SDL has a formally defined semantics
- But (to the best of my knowledge) there is no verification calculus (no proofs)
- Model-oriented specification
- Process specifications resemble flow diagrams $\Rightarrow$
  What about structured programming?
- Overall fairly low level of abstraction

Literature

- J. Ellsberger, D. Hogrefe, A. Sarma: **SDL - Formal Object-Oriented Language for Communicating Systems**, Prentice Hall (1997)
  Book on which this chapter is based. Informal description of SDL-96. Unfortunately, primary written for experts.
- D. Hogrefe: **Estelle, Lotos und SDL**, Springer-Verlag (1988)
  Introduction into three specification languages for telecommunication systems. Readable, but out-of-date.
- A. Sarma: **Introduction to SDL-92**, Computer Networks and ISDN Systems 28, 1603-1615 (1992)
  Introductory, short article on SDL.
- F. Belina, D. Hogrefe: **The CCITT-Specification and Description Language SDL**, Computer Networks and ISDN Systems 16, 311-341 (1988/89)
  A longer article on SDL-88.