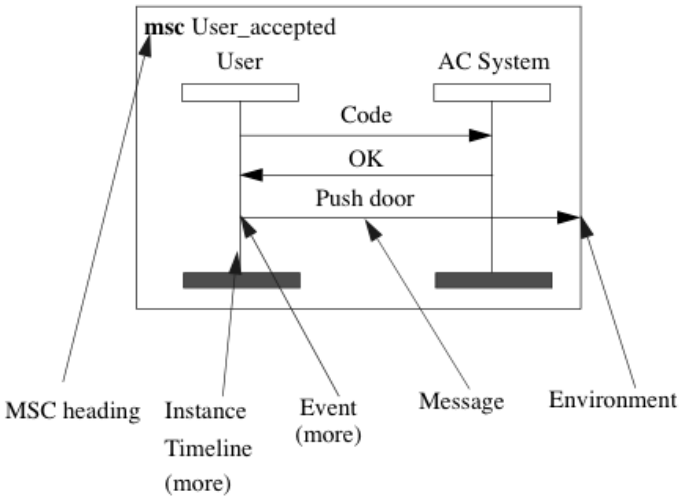
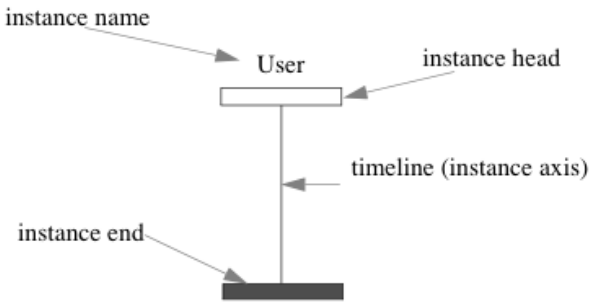
Sammendrag TTM4115

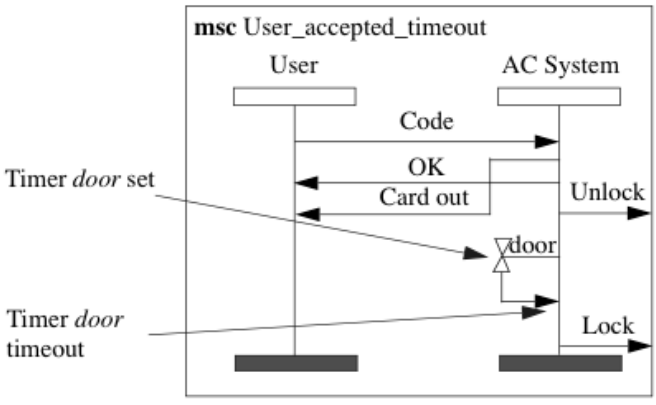
MSC: Interactions between the actors
SDL: Behaviour of each individual actor

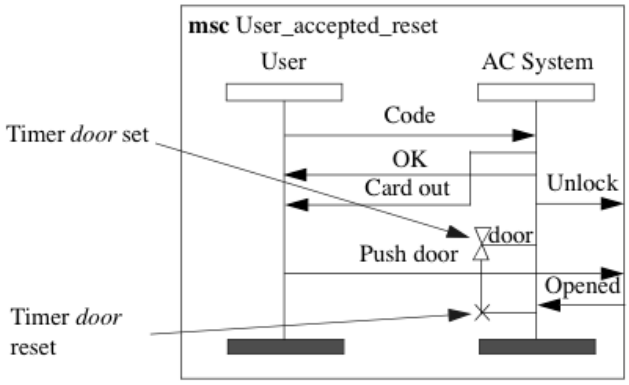
MSC

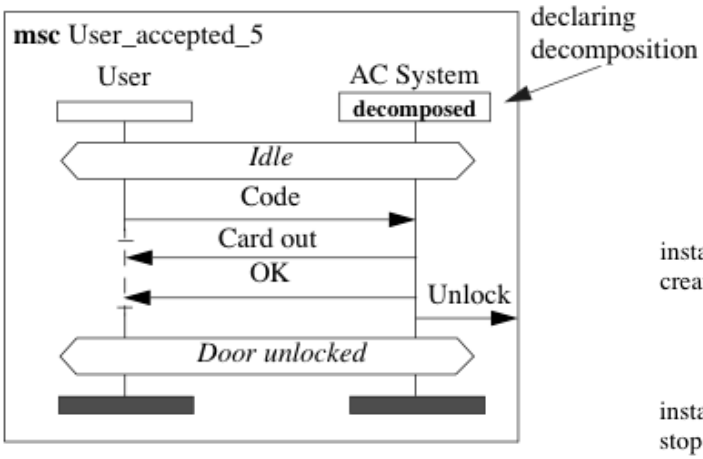
- Cannot describe absolute time.
- Asynchronous communication.
- The set of mscs used to describe a piece of reality is called an MSC document.
- Create alternatives by using conditions.
- MSC is formally defined using process algebra. Any chart in MSC can be transformed into a canonical representation.
- Rule of thumb: start with a few MSCs where the system is one instance, and the end users are either another instance or the environment.
- MSCs on high abstraction levels should not be crowded with all kinds of synchronizing messages or messages which serve only protocol purposes.



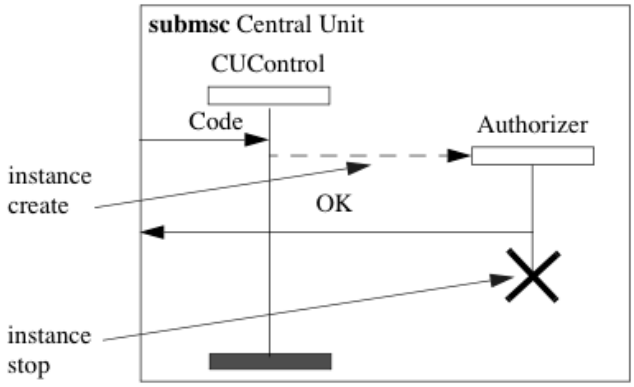


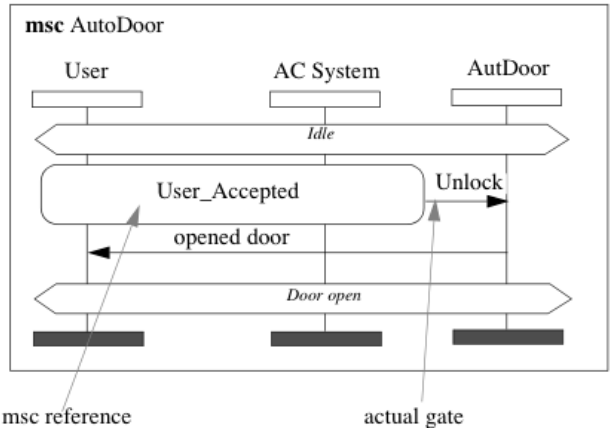


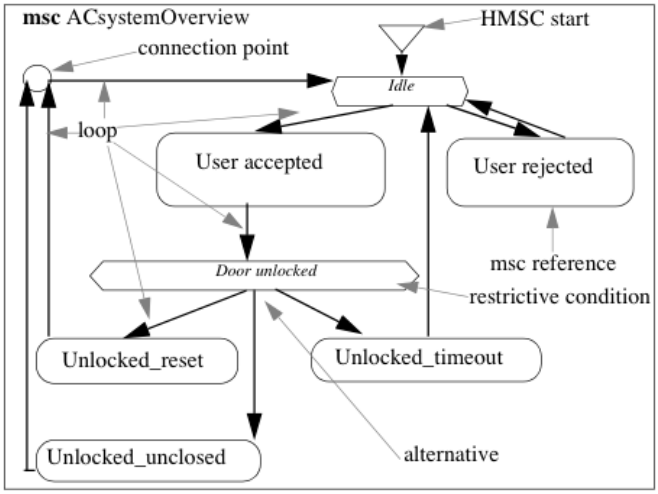




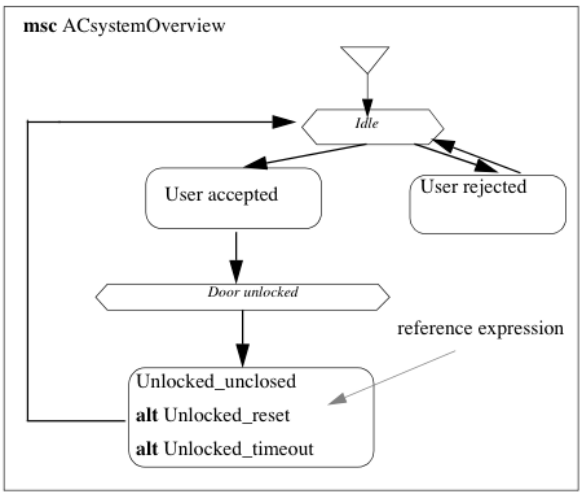
The AC System is then created with a **submsc** heading(removed in MSC-96)

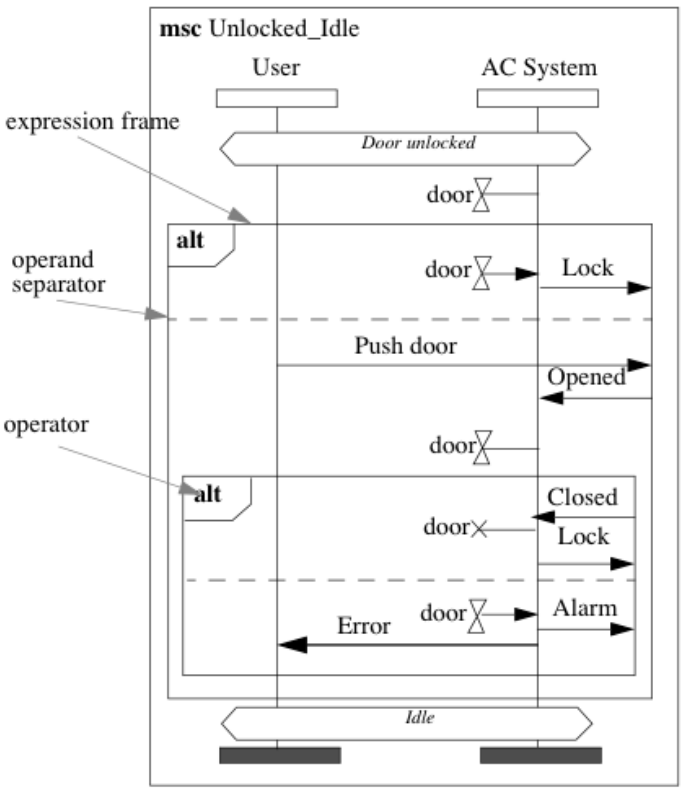


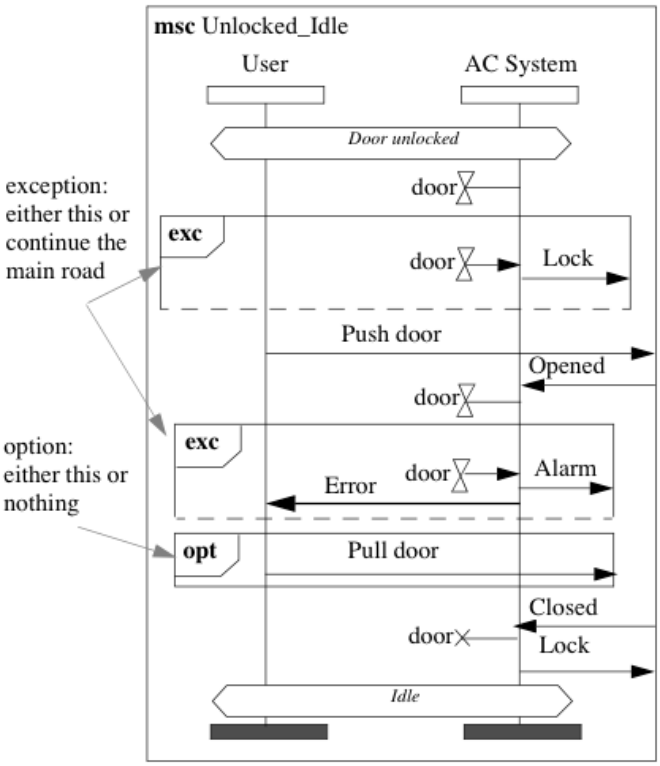


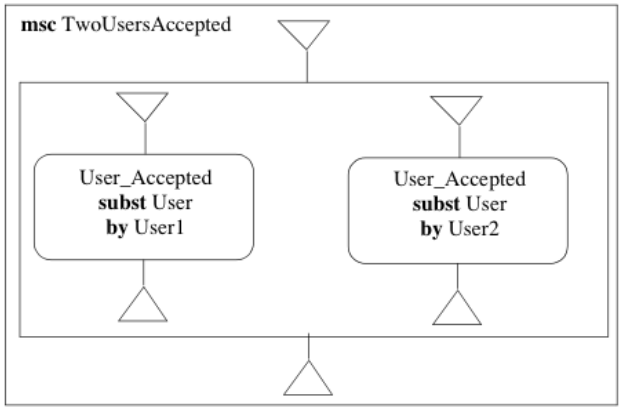


The restrictive conditions of HMSC apply only to global conditions.



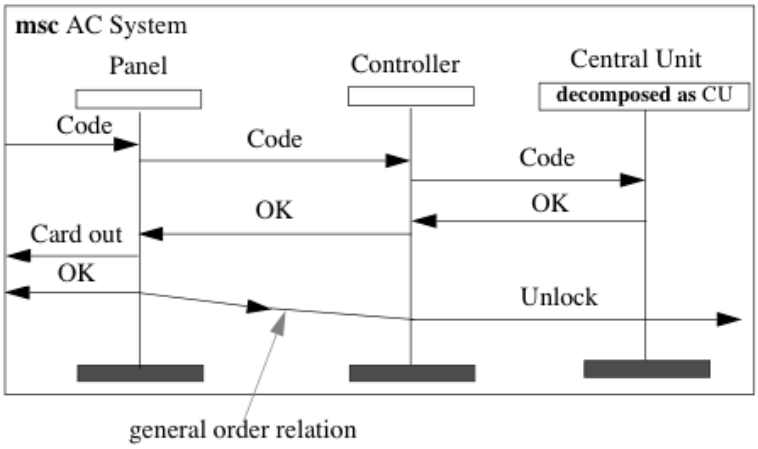






Parallel merge is used to describe situations which are independent of each other.

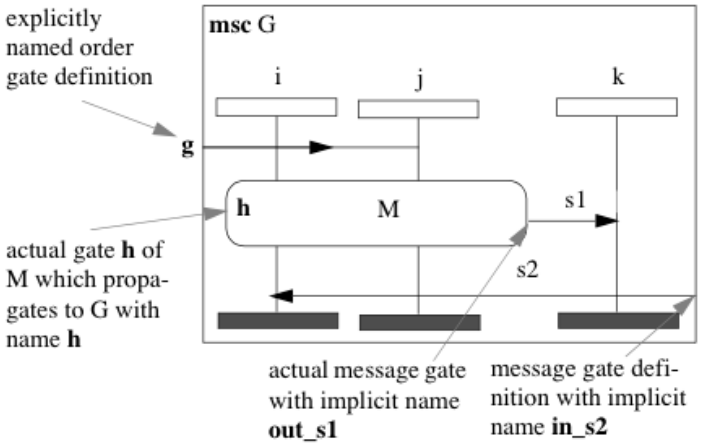
Substitution: Every MSC can be seen as a pattern where message names and instance names as well as MSC names can be exchanged.



The event at the beginning of the arrow must happen before the event at the end.

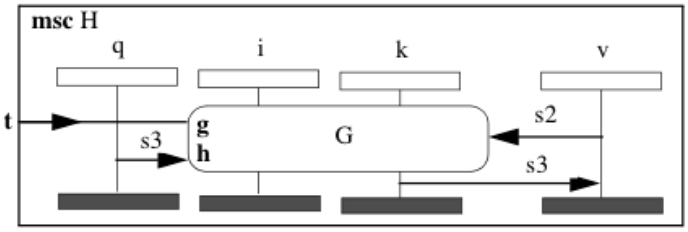
General order relations can end in a gate and thus events in one MSC can be ordered relative to an event in another MSC.

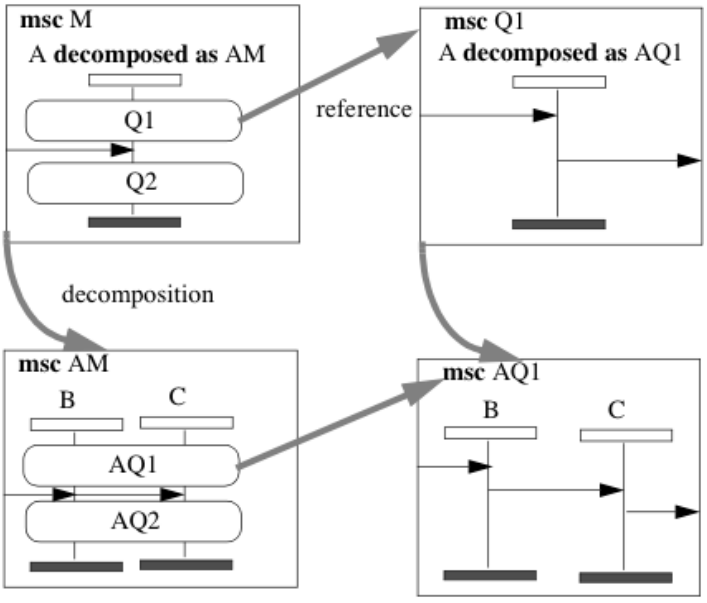
Never repeat instances in decomposition diagrams from upper level diagrams. Instead send messages to the environment.



The internal communication of M has no effect on the communication between M and its surroundings.

When a messenger boy turn up at gate out_s1, he will get an s1 message which he delivers to instance k.



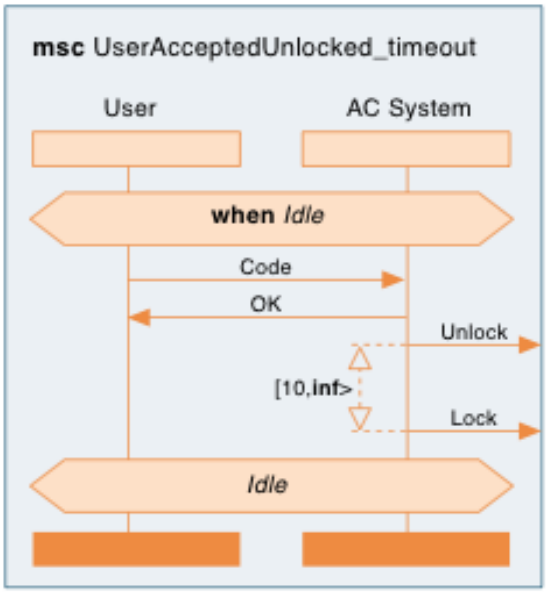


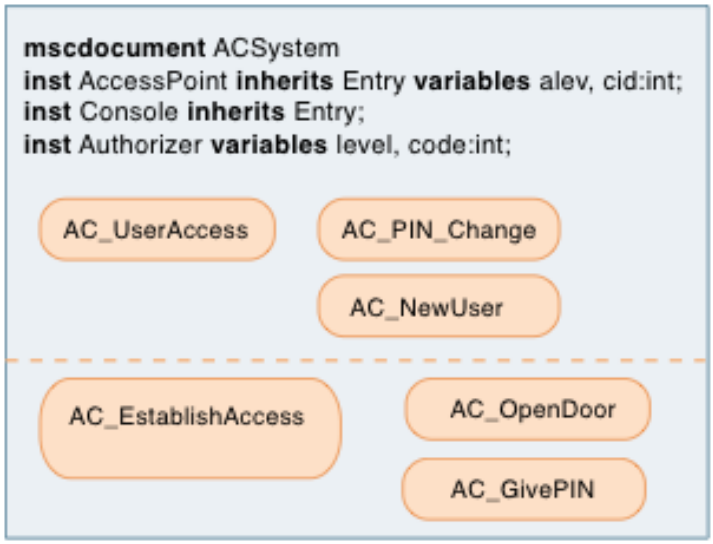
Whenever an instance which is covered by an MSC reference is decomposed, the decomposition should show the same structure of MSC references as the decomposed instance.

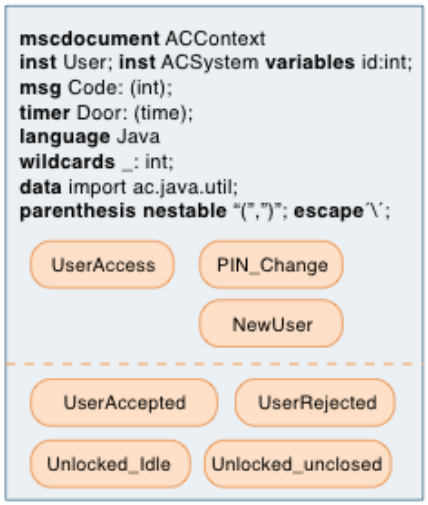
Global conditions of the decomposed instance should reapper in the decomposed MSCs.

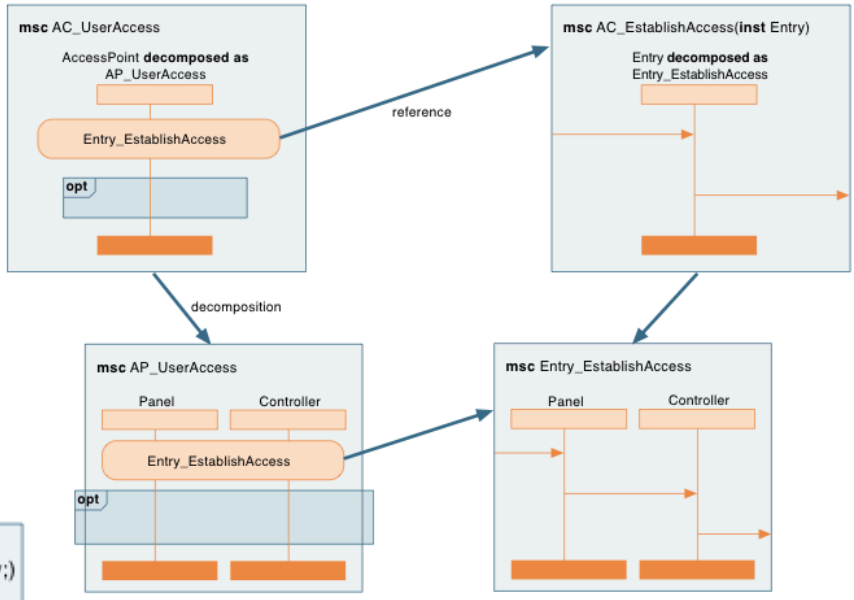
Methodological rules:

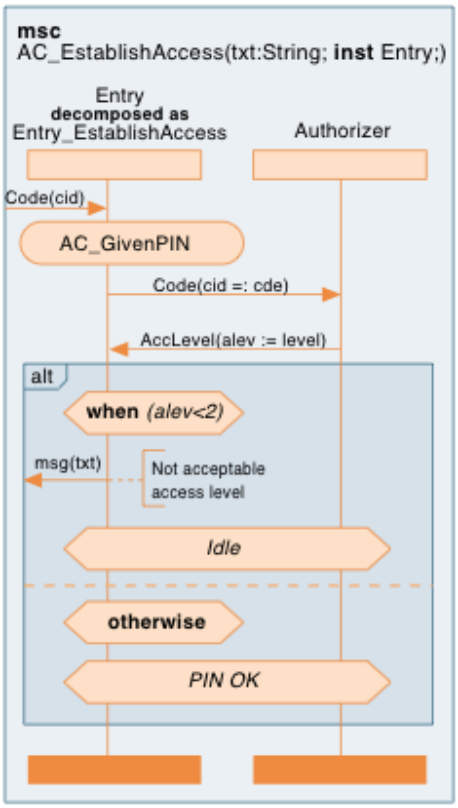
- Service orientation. Make one MSC per service.
- Role orientation. Instances of the MSCs in the domain are roles.
- Normal cases: Focus first on the normal cases and make them formal.
- Minimize informal text.
- Supplement the normal cases by cases expressing exceptional and erroneous situations.
- Use global conditions to describe important system states.
- Substitute complicated messages with MSC references.











Above the dashed separator are the defining MSCs and below are the utilities. Tenk på public vs private.

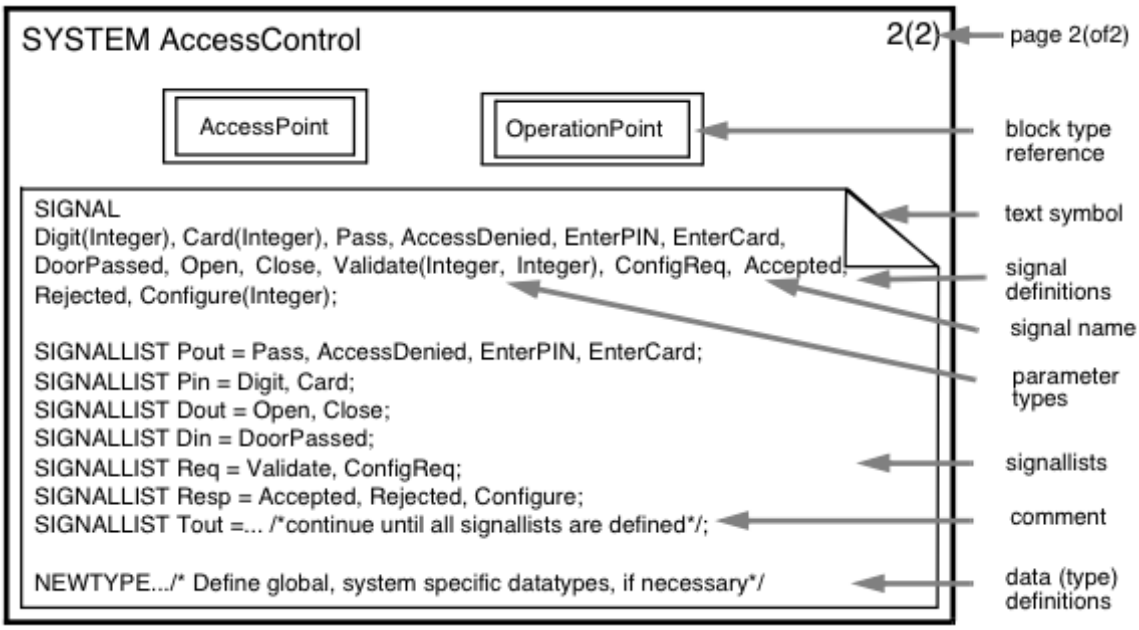
In AC_EstablishAccess AccessPoint have not been used directly, but rather Entry. This is acceptable as AccessPoint is a specialization of Entry.

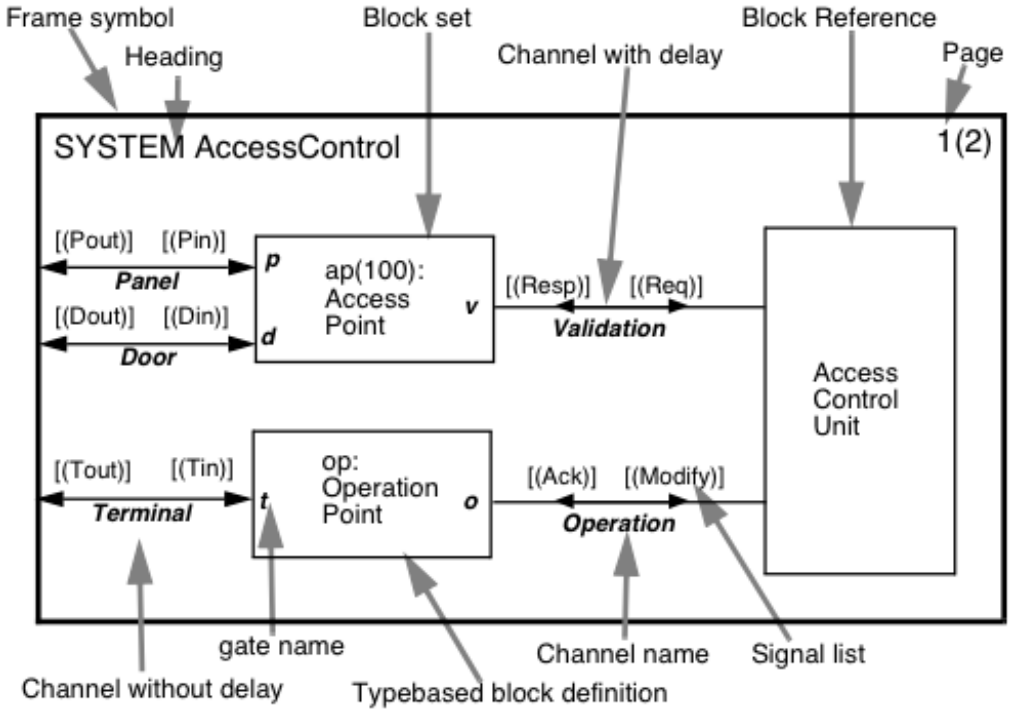
Static variables are given in the parameter list of the diagram. The dynamic variables are associated with an instance and defined in the MSC document.

Initial conditions of MSC diagrams are called *guards*. Their text is preceeded with the keyword **when**.

SDL

- For systems that are: reactive, concurrent, real-time, distributed & heterogeneous
- Composed of processes that behave concurrently with each other.
- The starting point is always a top level description of a system.



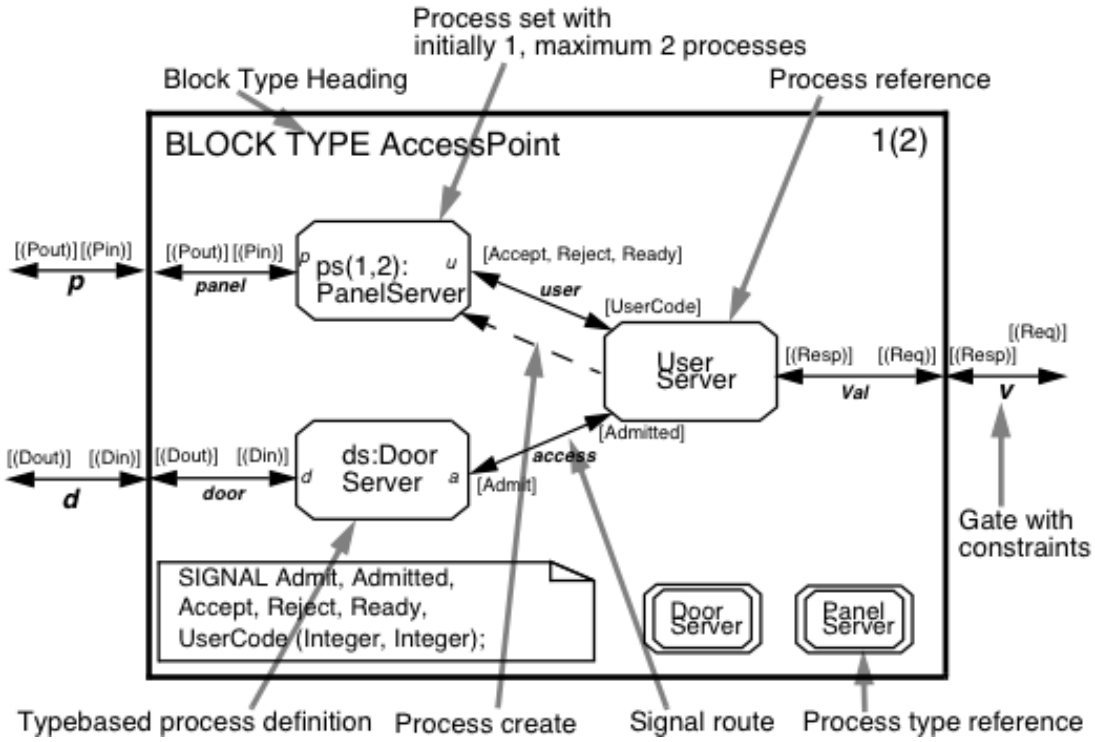


System

Outside the frame is the environment, which is not described.

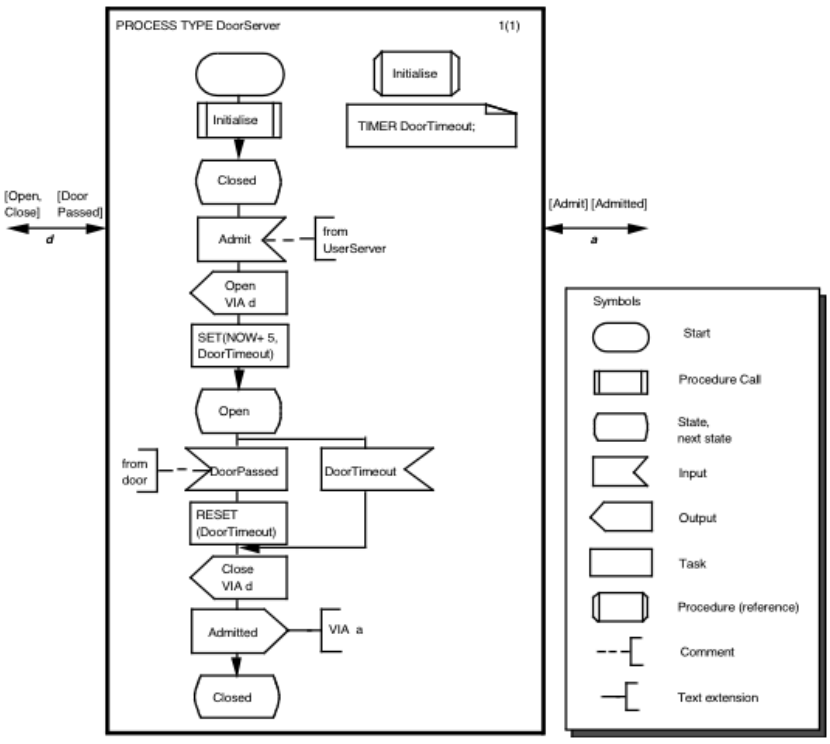
Blocks

- Each block contains at least one process.
- Blocks are containers, which are decomposed into blocks and channels, until the basic components, processes, are reached.
- Block types may contain data type definitions, but no variable declaration.

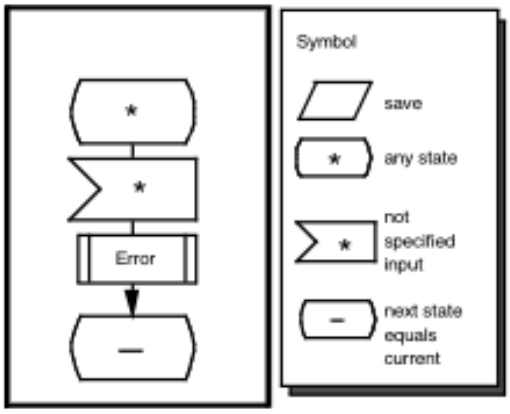


Process

- Communicate through concurrent signal routes and channels.
- Process = Extended finite state machines with timers.
- Each process consists of the input port, the FSM, timers and variables.
- For each signal it consumes, the FSM makes a transition from one state to another state.
- Use DCL to declare variables. SDL supports: Boolean, Character, String, Charstring, Integer, Natural, Real, Array, Powerset, and Struct for structures and user-defined types. Also supports Time, Duration, and the Process Identifier (PId).
- The acting unit of SDL. A unit of sequential behaviour.

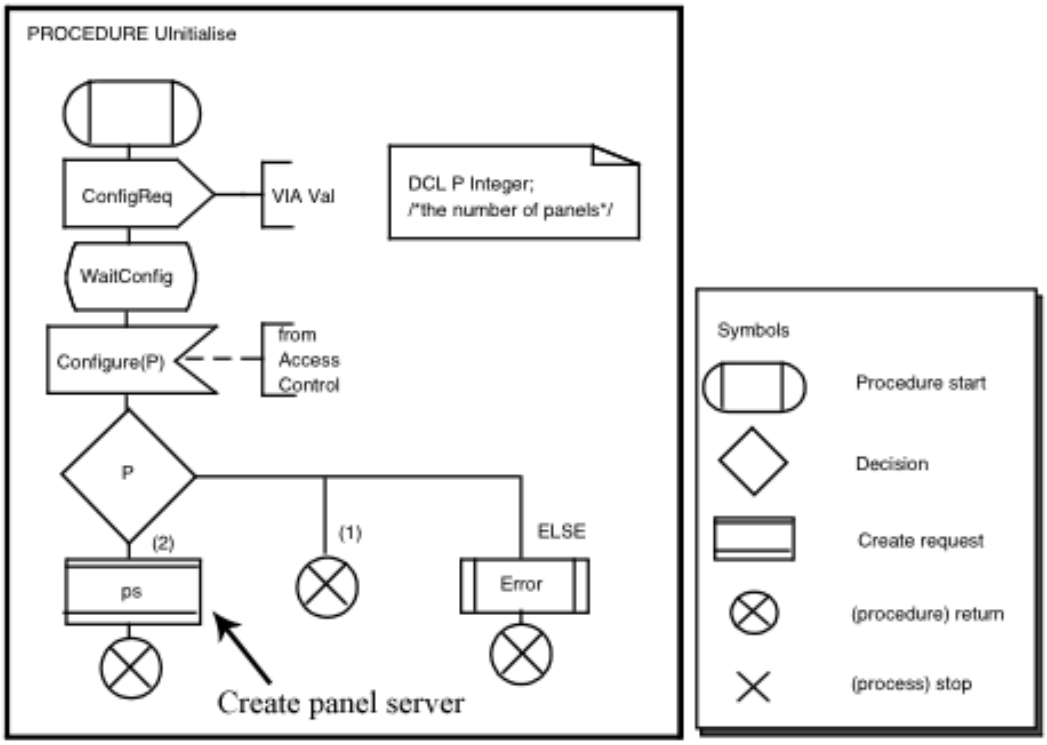


The essential **difference** between a block agent and a process agens is that the instances of agents within a block agent behave concurrently and asynchronously with each other, whereas instances within a process are scheduled one at a time.



Signals that are saved will be kept in the input port in the order of their arrival until a new state is reached.

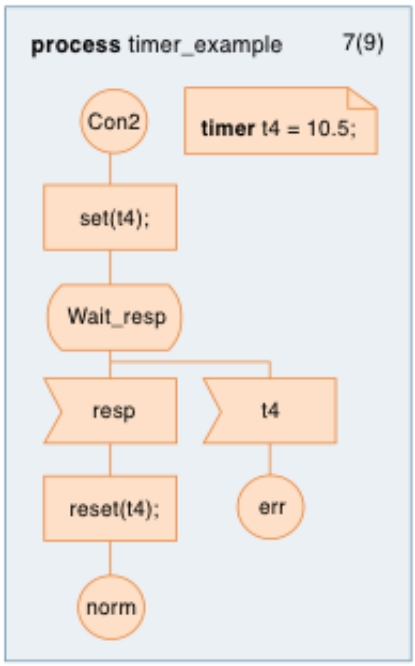
Forklaring for figuren til venstre: for every state, if unspecified consumptions occurs, the procedure Error is invoked, and the next state remains the same as the current state.



- Blocks cannot be created dynamically.
- Only members of process sets where the maximum number of instances have not been reached may be created.
- PId expressions: self, parent, offspring, sender

Timers

- *active(t4)* tests if the timer *t4* is active.
- *set(now+3, t4)* sets the timer to 3 from the current time.
- *set(t4)* sets the timer *t4* to the duration given in the timer definition.
- *reset(t4)*
- If the timer expires then a signal of the same name is put in the input port of the agent.
- Timer definitions are NOT allowed in state diagrams or procedure diagrams.



- If several objects have the same properties, using explicit types makes the SDL simpler. In addition its properties can be inherited to make specializations of the type.
- When a type simply inherits from another type, it has the same set of properties as the original type, but a distinct identity.
- A redefined item is virtual, and ban be redefined again if the subclass is inherited again.
- When redefinition is given, an item can be made finalized, so it cannot be changed in subclasses.
- Symbols with dashed lines indicate the use of existing items defined in a super type.
- Specialization of types can also be done using context parameters. Formal context parameters are given in a type definition after the name of the type and enclosed in < >. Can be a signal, a block, a process, a data variable, a synonym, a gate, an interface, a procedure, an exception, a timer, or a type.
- Context parameters, virtual types and gates can have constraints.
- A package groups several type definitions together and allows them to be used in several systems.
- Kan kun redefinere dersom orginalen er virtual.
- If no properties are included in the three comparment class symbol, it can *iconized*.

Data

- SDL data is strongly typed. Can either be a value type (represent set of values) or an object type (object reference).
- User-named types are defined either using Predefined types with parameters or by constructing new data types. A **struct**ure has any number of fields, each of which can be any named type including other structures, string, vectors or arrays.

```
value type S { struct
    a   Integer;
    b   Charstring optional;
    c   Character default 'd';}
dcl s1 S, I Integer, X Character;
    s1:=(.3,'21','e'.);          /*structure value*/
    s1.b:=mkstring(s1.c);        /*field access*/
```

- A choice is similar to a structure, but can only contain one field at any one time, and assigning one of the choices makes all other choices undefined.

```
value type C { choice
    hue     rgb;
    bs      Bitstring;}
```

- A synonym type can be defined for any data type. Usually combined with some limitation of the values of the parent type.

```
syntype Int16 = Integer constants 0:65535;
```

Agent Creation

- The definition of an agent include how many initial instances of the agent there will be, and the maximum number of instances. Default is one initial, infinite maximum.
- Agents can be created by other agents in a create request as part of a transition.

State machine diagrams

- The state machine diagrams determine the behaviour of Agents. They define what happens in each state and the transitions between states.

Procedures

- Similar to a state machine diagram except that it starts with a *procedure start* and ends with a *return*. Provides a level of abstractions and a component for reuse.

Combined with UML

- An SDL System consists of Agents that are connected by means of Channels. Agents may communicate by sending Signals or by requesting other Agents to perform Procedures. An Agent may have both a StateMachine and an internal structure of Agents. The internal Agents and the StateMachine are connected by Channels. The connection points for Channels are Gates.

- Types correspond to classes in UML.
- Block Agents are concurrent Agents with possibly interleaved execution of the transitions of the state machines, while the Process Agents are alternating Agents with run-to-completion execution of transitions.
- An Interface defines Signal, Variables, RemoteProcedures and Exceptions. Interfaces are associated with Gates. Gates are connection points for Channels connecting Agents. Communication between Agents takes place via Channels.

A-rules (Analysis)

Guidelines for system analysis. (s74, bok)

- **Problem statement**. Make a statement that explains the problem domain. Focus on the purpose, the essential concepts, the procedures and the rules. Normally sufficiently expressed using natural language.
- **Dictionary**. Make or obtain a dictionary for the problem domain. The dictionary should be kept updated throughout the development.
- **Concept model**. Make a static conceptual description of the problem domain.
- **Specialization hierarchies**. For each concept in the dictionary, ask whether all the objects that fall within the extension of the concept have the same properties.
- **Context**. Make a context diagram where the system is identified and the system environment is detailed. Describe communication interfaces and other relations the system shall handle.
- **MSC**. Make MSCs that describe the typical interaction sequences (protocols) at each layer of the interfaces.
- **Role behaviour**. Define the interface behaviour of each role in the system and in the environment. Use the roles as basis for behaviour synthesis and validation.
- **Sketch system structure**.

S-rules (Structuring)

- **State orientation**. Represent what the environment may distinguish as control states of the process, as states in the process graph.
- **Decisions**. Critically review all decisions to ensure that they are not symptoms of undesirable state hiding.
- **Signal set**. Represent what the environment may distinguish as different control signals by different signal types.
- **Control flow**. Branch on input signals rather than on decisions.
- **Data**.
  - When the process graph structure is not dependent on the data values.
  - To keep information of the situation and structure of environment.
  - To control loops that are not terminated by specific signals.
- **PId variables**. Use a PId variable to represent each process role in the environment of a process type. Give the variable the same name as the role.
- **Concurrency**. Model independent and parallel behaviours as separate processes.
- **Golden rule of partitioning**. The golden rule is to partition along the lines of independence and not across dependencies. A system should be decomposed into modules with low external coupling and strong internal cohesion.
- **Similar process instances**. Do not hide similar behaviours in data. Decompose such that similar independent behaviours are described explicitely as separate instances of a process type.
- **Interaction processes**. Use one process to play each independent behaviour role required by the environment.
- **Inter-conncetions**. Use one channel and/or signal route to carry each independent and concurrent interaction dialogue.

- **Protocol layering**. Protocols should be decomposed by layering, such that each layer hides the details of the protocols used on that layer from higher layers. Lower layers should provide application independent transfer services for the upper layers.
- **Shared data**. Introduce special processes to encapsulate shared data. Encapsulate data needing independent access in separate processes.
- **Resource allocation**. Introduce a special resource allocator process to control the access to each pool of functionally equivalent resources.
- **Procedures**. Model independent sub-behaviours within a process as procedures. Look for sub-behaviours that can be perceived as one operation at the calling level.
- **Similarities within processes**. Use procedures within processes to single out patterns that recur in the process.
- **Macros**. Never use macros where procedures are applicable.
- **Services**. May be applied if:
  - the process is naturally divided into quasi-parallell sub-behaviours having separate state spaces.
  - the decision of which service to apply is given uniquely by the input signal type.
- **Data types**.
  - Use pre-defined data types whenever applicable.
  - Use pre-defined generators to make arrays, sequences and sets.
  - Use STRUCT to make more advanced data types.
- **System and Environment**. For the elements at the periphery of your concern, place them inside the system if you will describe their behaviour in detail.
- **Block purposes**. Gradual approach to detail; Units of reuse and repitition; Encapsulation of layering; Encapsulation of independent adaptation and change; Limited scope of process creation and communication; and Correspondance with the physical system.
- **Steps to derive a block partitioning**.
  - Partition the significant environment behaviour into nested blocks in a way that fulfils the desired block purposes.
  - Identify nested behaviour roles that blocks in the environment give to the system.
  - Mirror the behaviour roles by actors in the system.
  - Make a conceptual description of context knowledge needed by the system. Allocate the knowledge to blocks and processes in the system.
  - Identify shared resources and introduce corresponding resource allocation processes.
- **Control processes**. Let leaf blocks have
  - one process for each independent communnication with its context
  - one process for each pool of shared resources to be dynamically allocated
  - one process for each block of shared data accessed and controlled independently
- **Routing processes**. For each block where there is a choice between local and remote communication, use a two-level addressing scheme, and hide the routing knowledge in a routing process. (s. 214)
- **Similarity**. Look for similarities, which will make type concepts. Re-examine components that are partially similar, but partially dissimilar.
- **Input consistent process**. Design SDL processes such that all their role behaviours, i.e. the behaviour visible to processes in their environment, are strongly input consistent. (s 354)

N-rules (Notation)

- **Names in macros**. Let each macro have a unique number as parameter, and let all the varying names that are visible at the macro border include the number.

- **State sub-diagrams**. Consider one state and the transitions from it as a sub-diagram which is described in one place.
- **Source and Destination**. Specify the sender of an input signal by means of a comment symbol, and the receiver of an output by means of a text extension symbol.

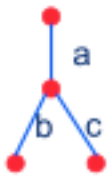
Mer om SDL

- SDL systems contain concurrent processes. Concurrency is an ideal model of independent behaviour.
- Blocks are not described as a FSM. Eventually contains processes.
- The common predefined types are: boolean, character, charstring, integer, natural and real.
- A macro is a pattern which is replicated where the macro calls are.

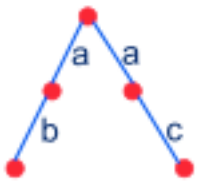
Process algebra

; is the *sequence* operator
+ is the *choice* operator
| is the parallell operator
\ is the restriction operator

S3 = a (b + c)



S4 = a b + a c



S3 and *S4* are *trace* equivalent: *{(a;b), (a;c)}*
But they are not observation equivalent: *S3* ¬ ≈ *S4*
S3 can always do both b and c after a, while S4 can sometimes do b and sometimes c.

S0 = a S1

S1 = b + g S3

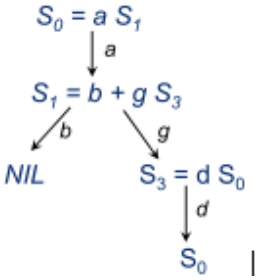
S3 = d S0

S0 = a (b + g d S0)

RST = Rigid synchronization tree
LTS = Labelled transition system

Common to define the semantics of process algebras in terms of LTS. These are defined in terms of a set of states and transitions between states.

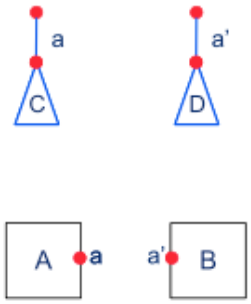
$S_0 = a\ S_1$
$S_1 = b + g\ S_3$
$S_3 = d\ S_0$



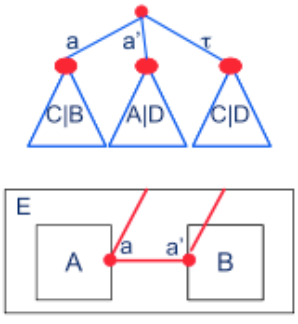
E = A | B: either A does an action alone, or B does an action alone or they do an interaction together.

A = a C

B = a' D



E = A | B
= a (C | B) + a' (A | D) + τ (C | D)



Composed in parallel interaction is possible between complementary actions.

For the parallel composition *E = A | B* this gives three possible first transitions: either A performs the a action alone; or B performs the a' action alone; or they perform an interaction together, represented by the action τ.

After *A* has performed the *a* action alone, the remaining behavior is *C | B*. This is expanded in the same way. Similarly after *B* has performed the *a'* action alone, the remaining behavior is *A|D*. Finally, after the interaction, the remaining behavior is *C | D*.

Remove as many τ-transitions as possible.

In the expression $A|B\backslash a$ the action *a* and its co-action *a'* are hidden from the environment, so that it is no longer possible to do *a* or *a'* actions alone in $A|B$.

Hide all actions where interaction is possible: $A||B$

A situation where no further action is possible even though the composed behaviors are not empty is called a *deadlock*.

Writing $B[b/a, c/d]$ means that action *a* is renamed to action *b*, and that action *d* is renamed to action *c* in B.