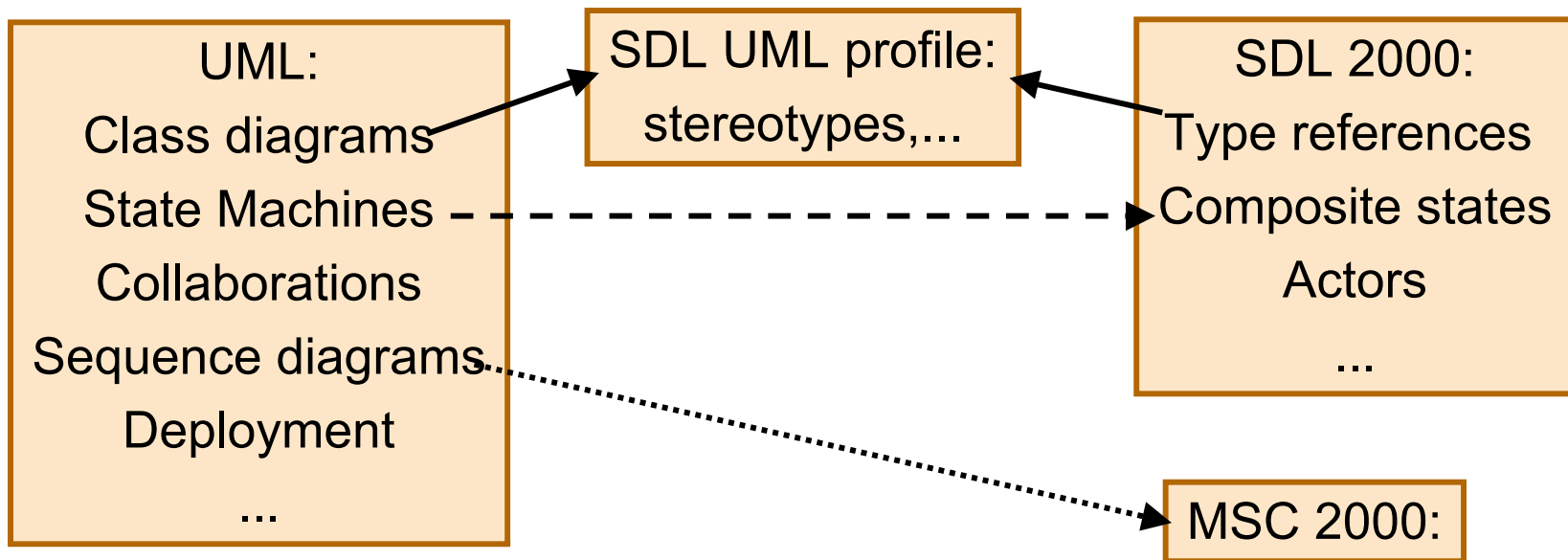


SDL-2000 = SDL-96 + UML + -

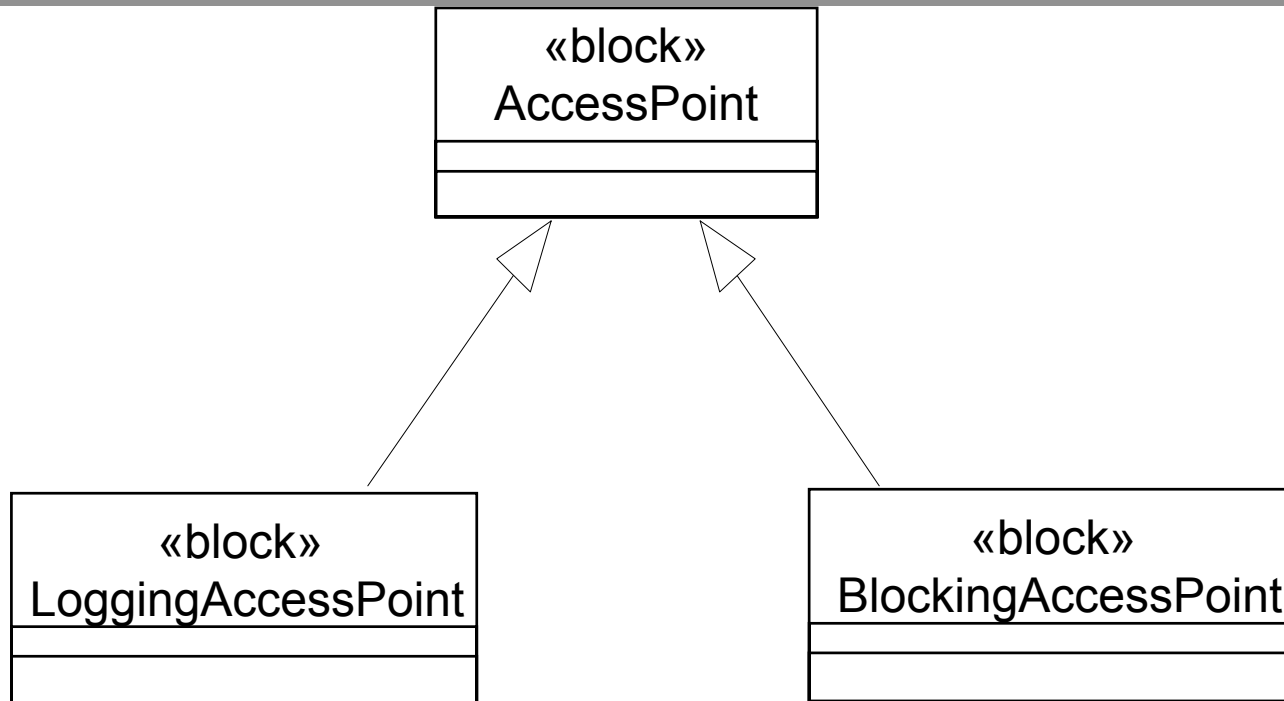


New ITU-T SG10 recommendations, due November 1999:

- **SDL-2000 (Z.100)**
- **SDL combined with UML (SDL UML Profile - Z.109)**
- **MSC-2000 (Z.120)**

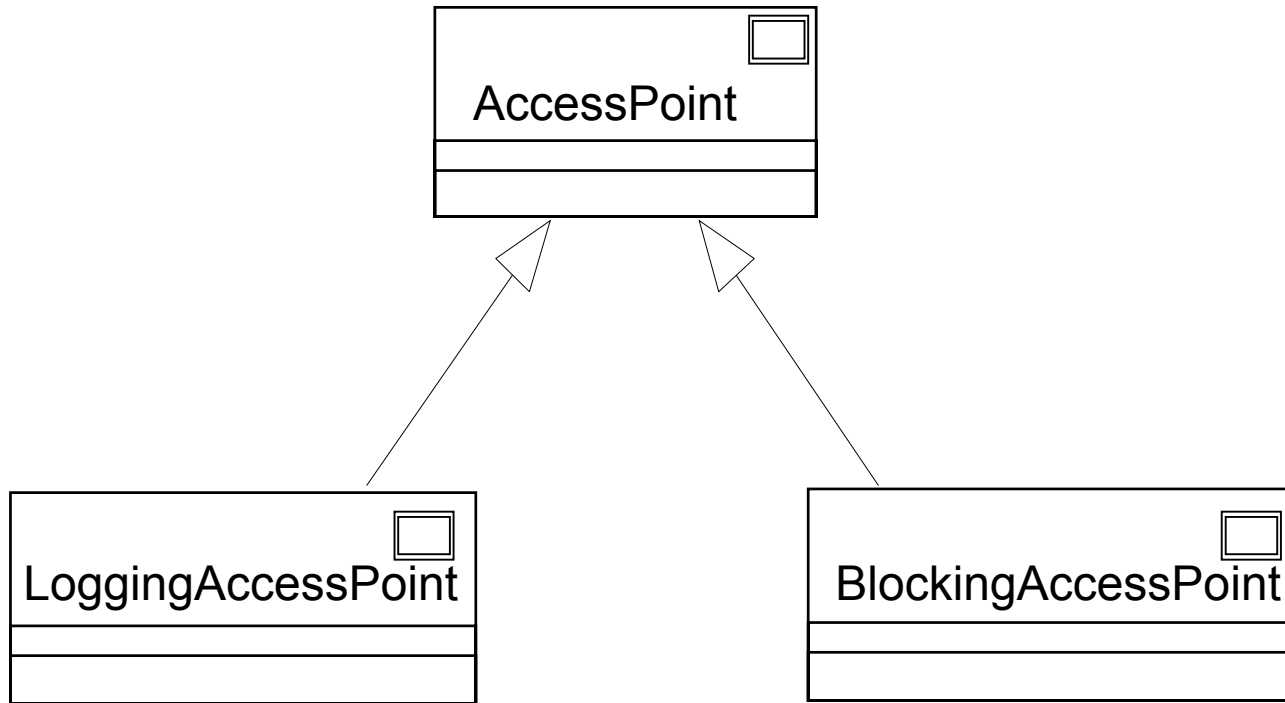
Using the UML SDL Profile . . .

- with stereotyped classes
- ... and associations, in this case specialisation

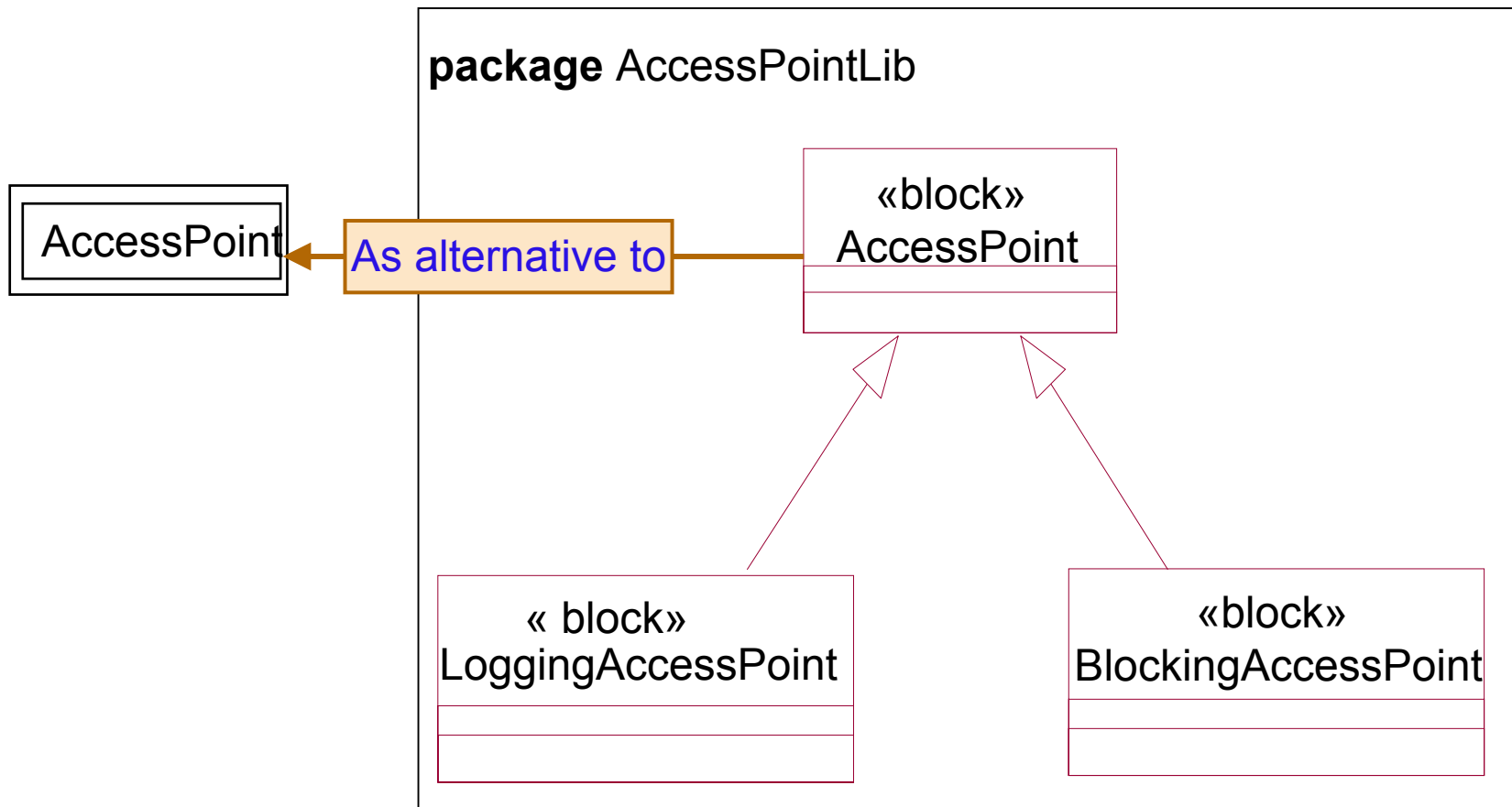


• • •

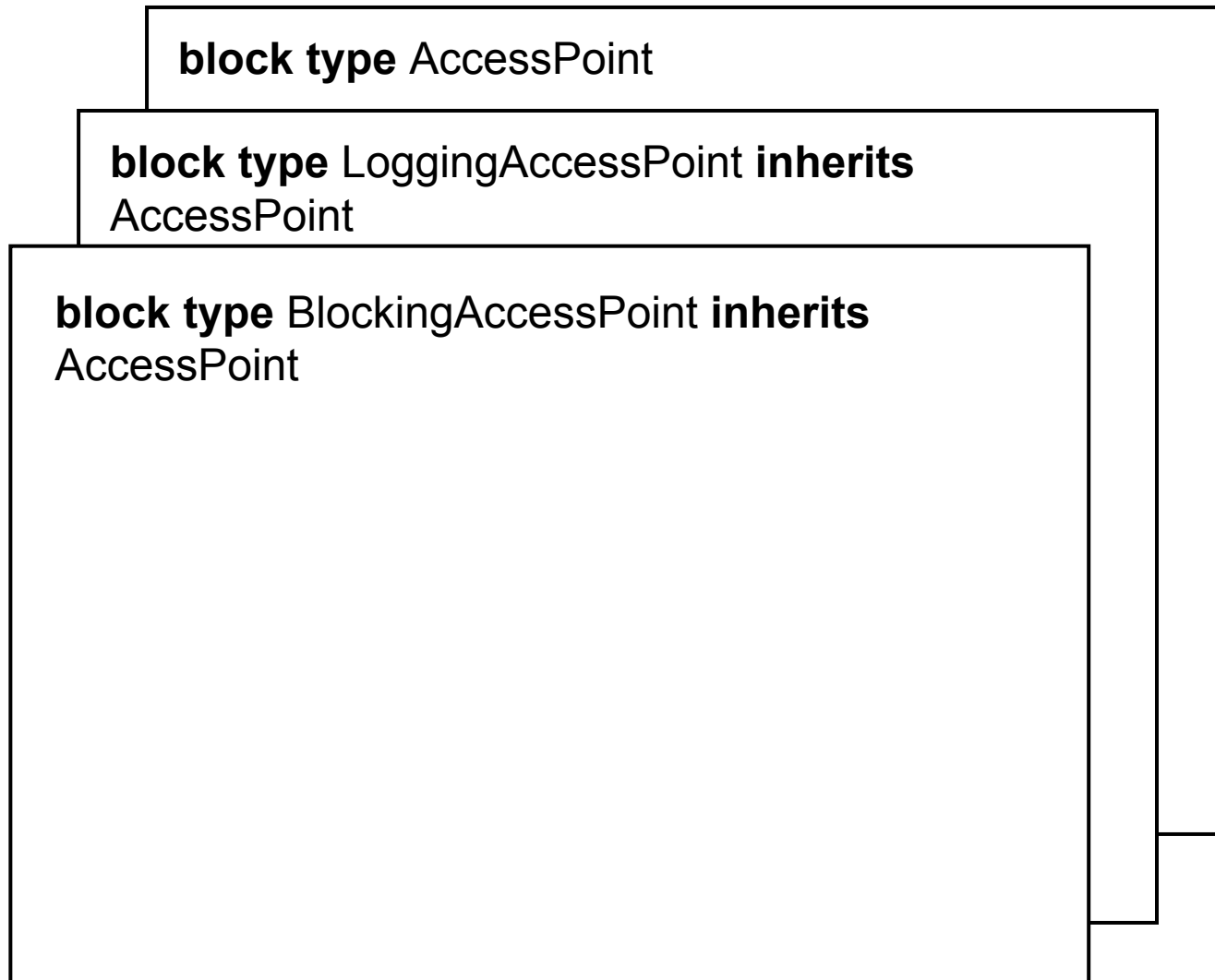
- or with the graphical alternative to stereotypes



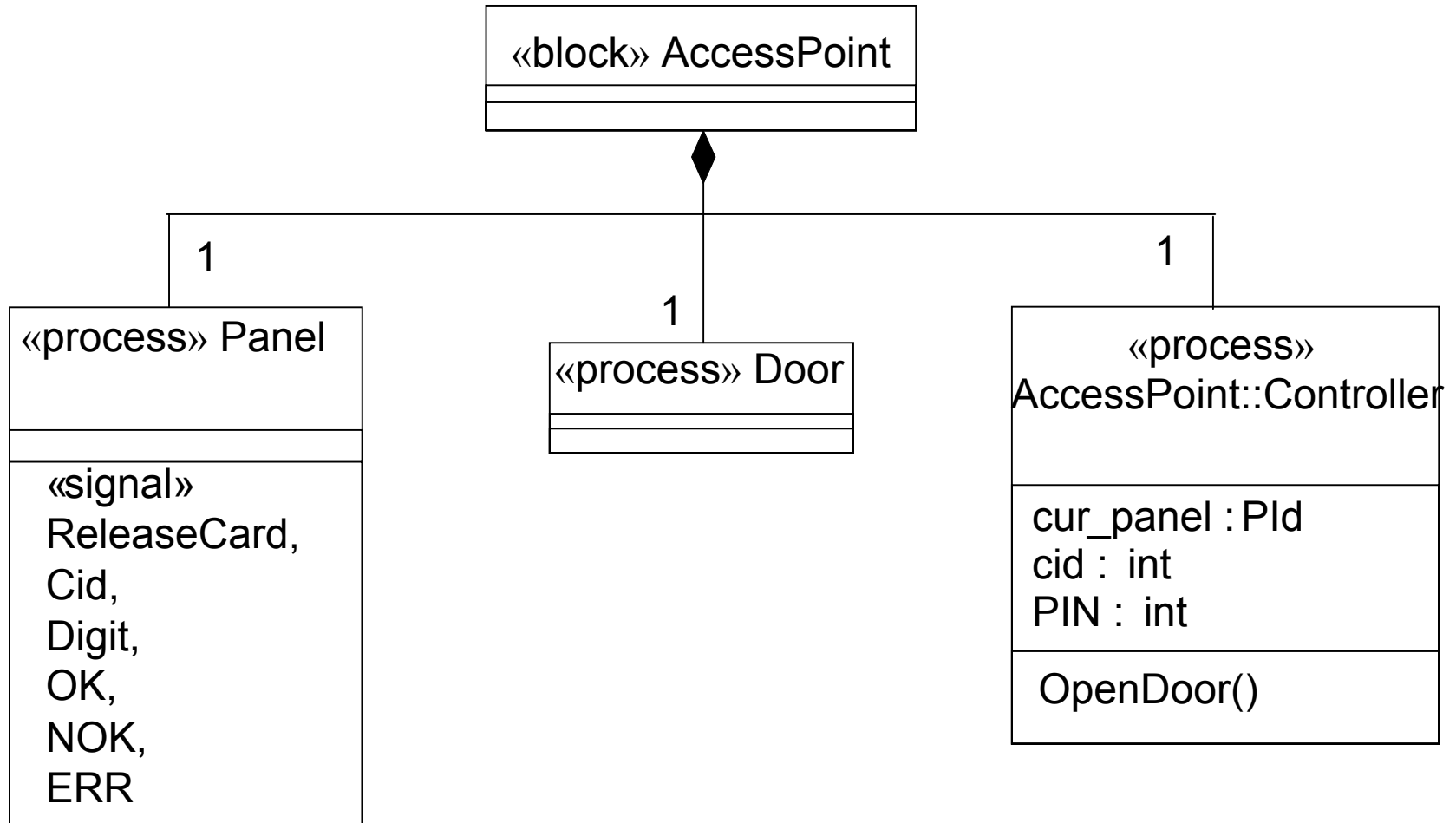
. . . part of the SDL model has been specified



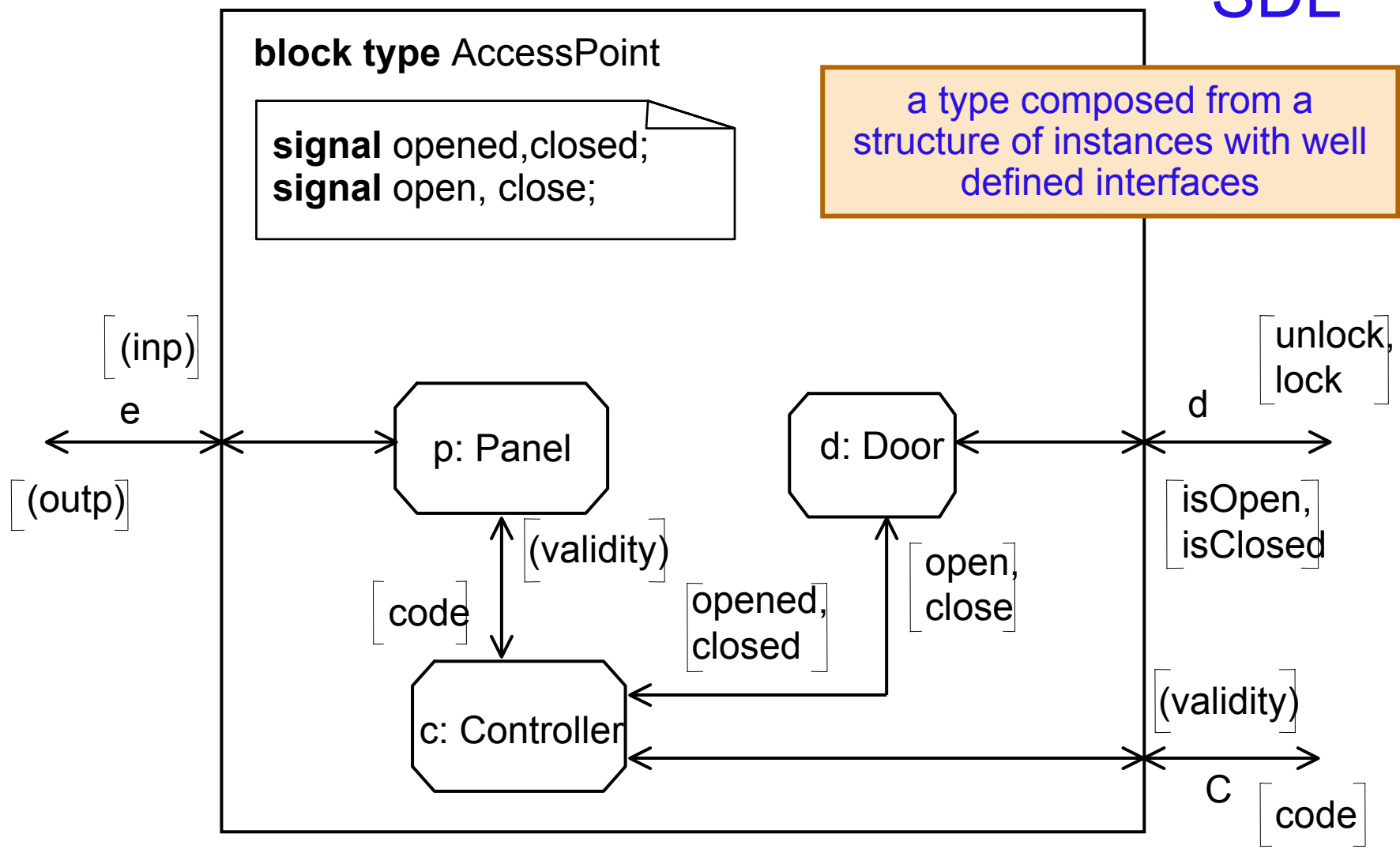
Including the type diagrams



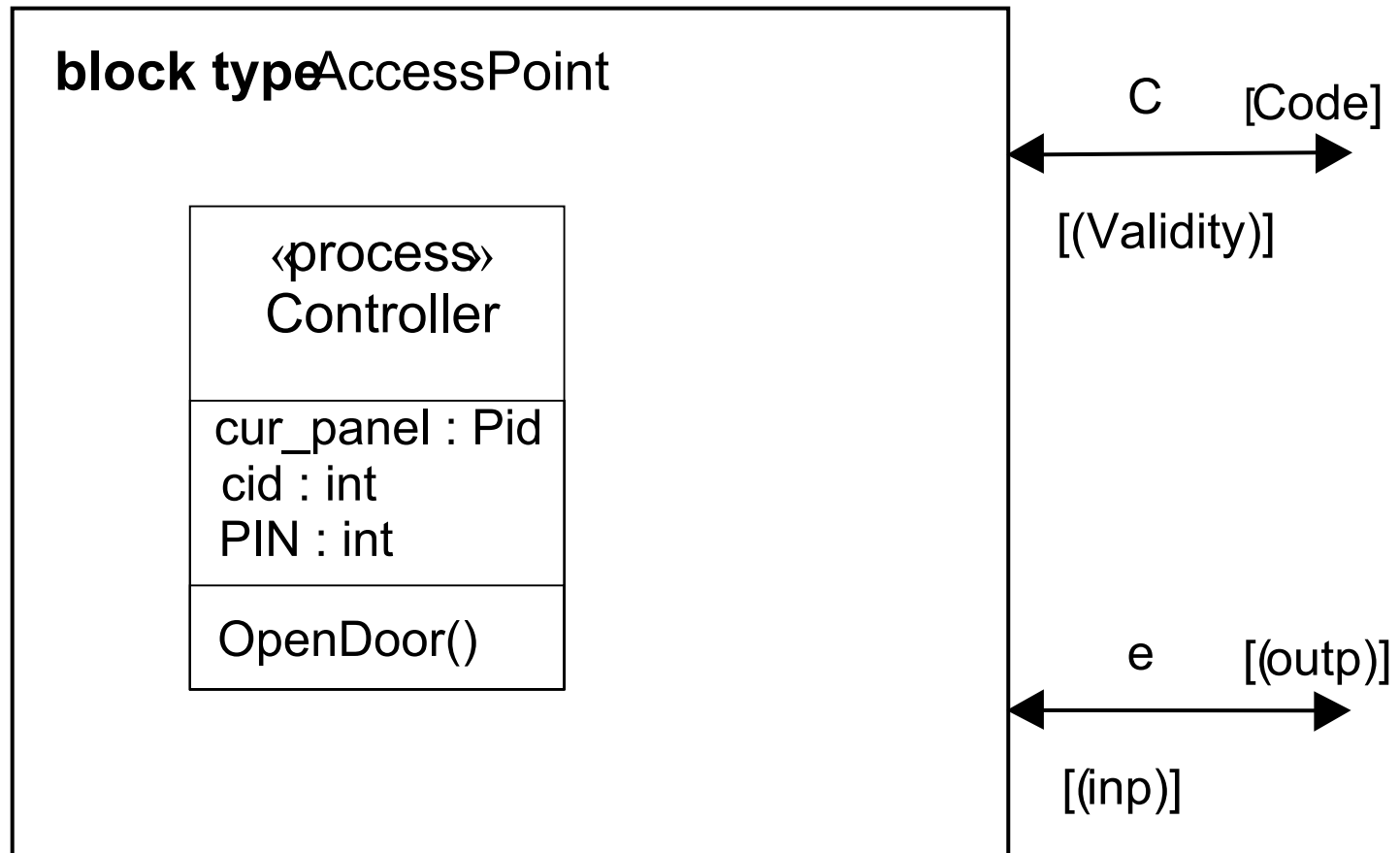
Composition specified on types . . .



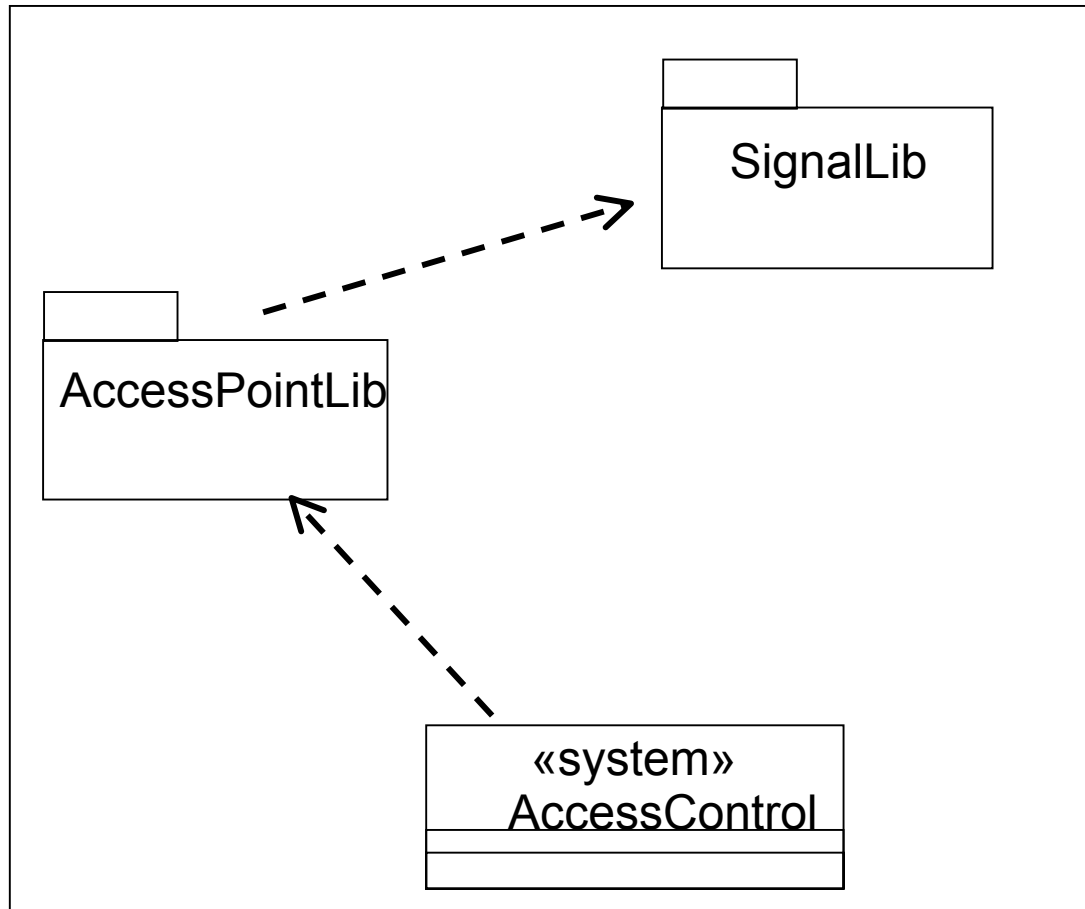
... is a partial description of this more detailed **SDL**



Not only type references, but also partial type definitions

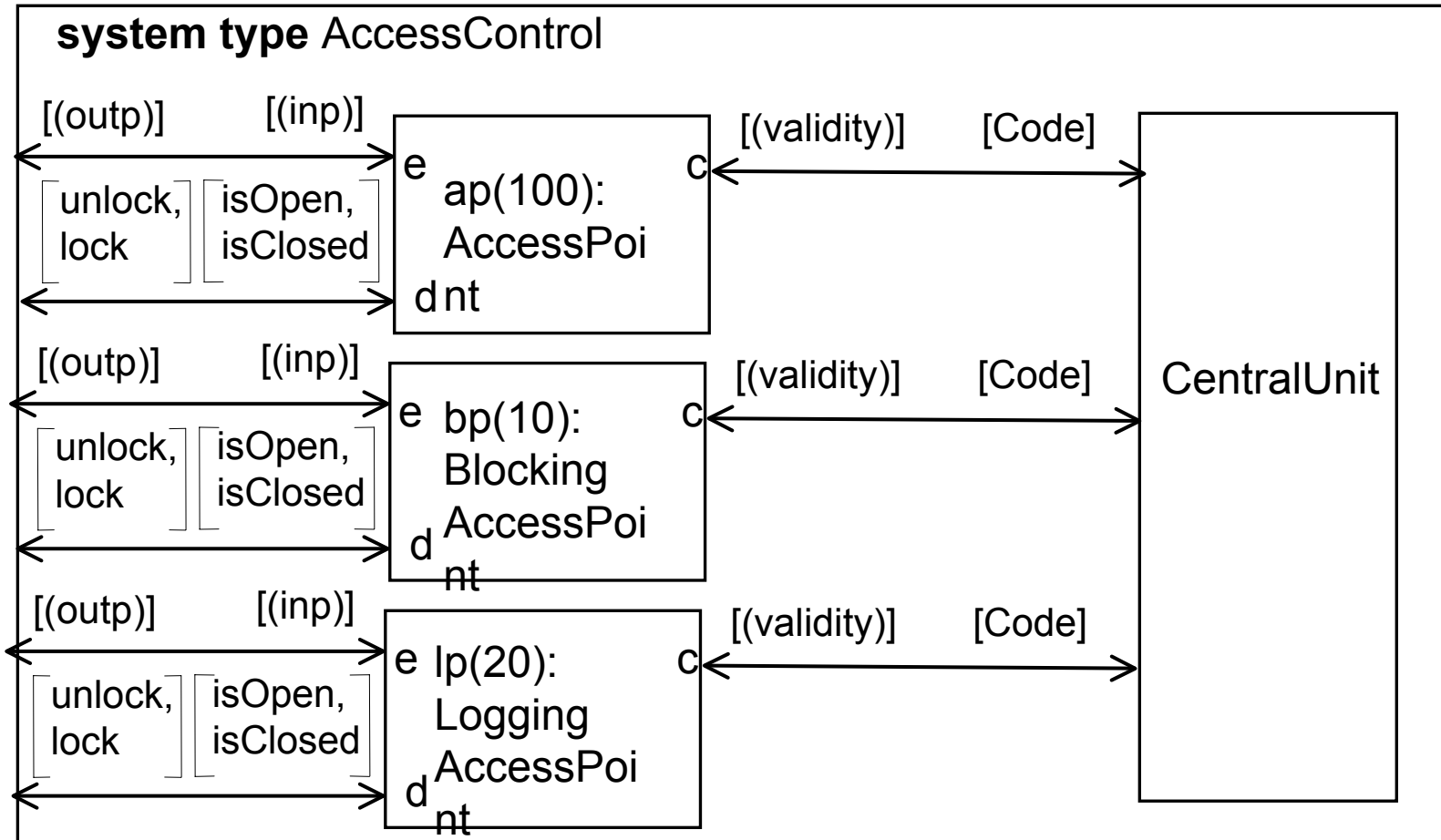


Use dependencies between packages and systems



The corresponding package reference associated with each diagram in SDL

use AccessPointLib



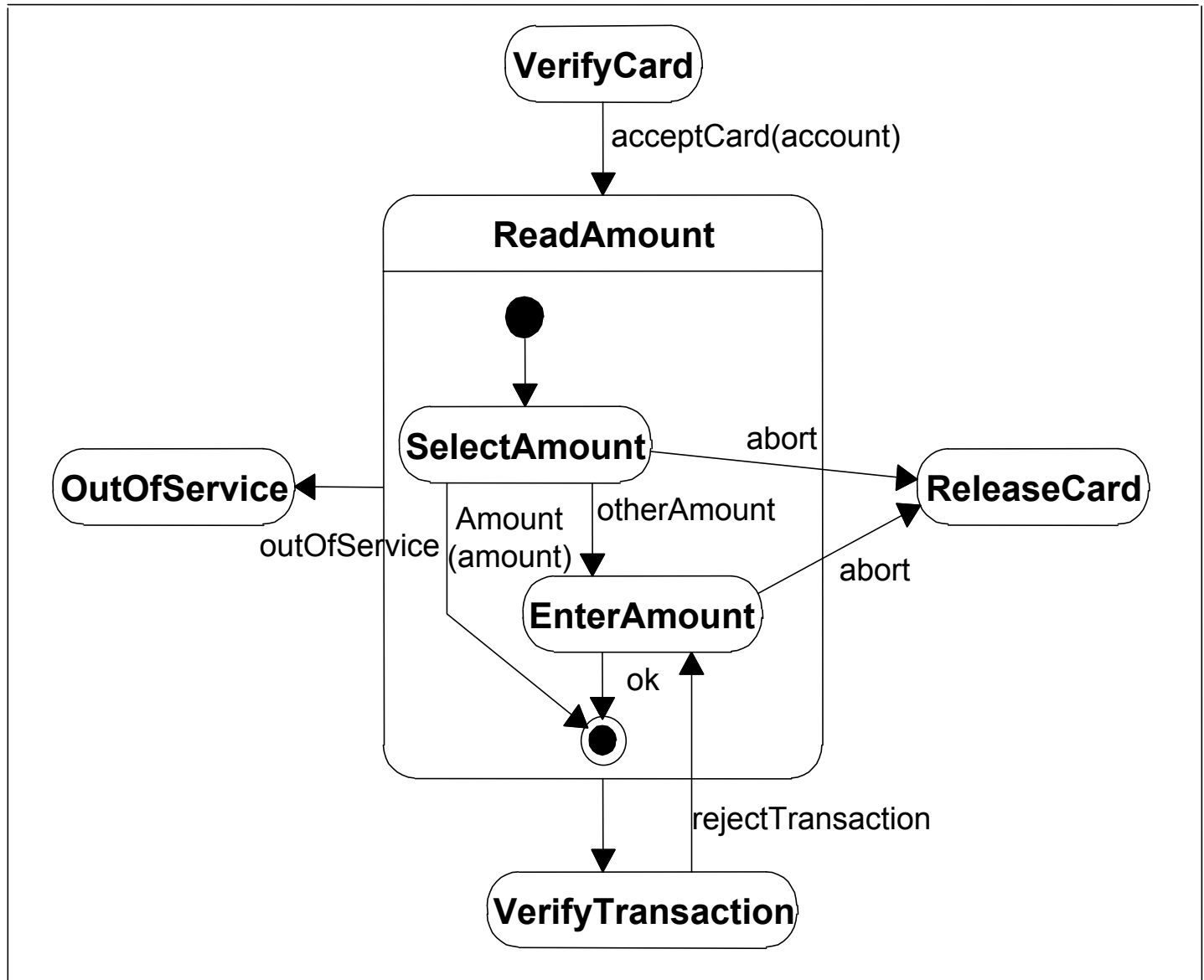
Composite states

- States with states and transitions, separate state diagrams, entry/exit points
- state overview diagrams
- state types and subtypes
- virtual states
- parameterized state types

- Combining
 - *State-orientation of Statecharts*
 - *Transition-orientation of today's SDL*

UML

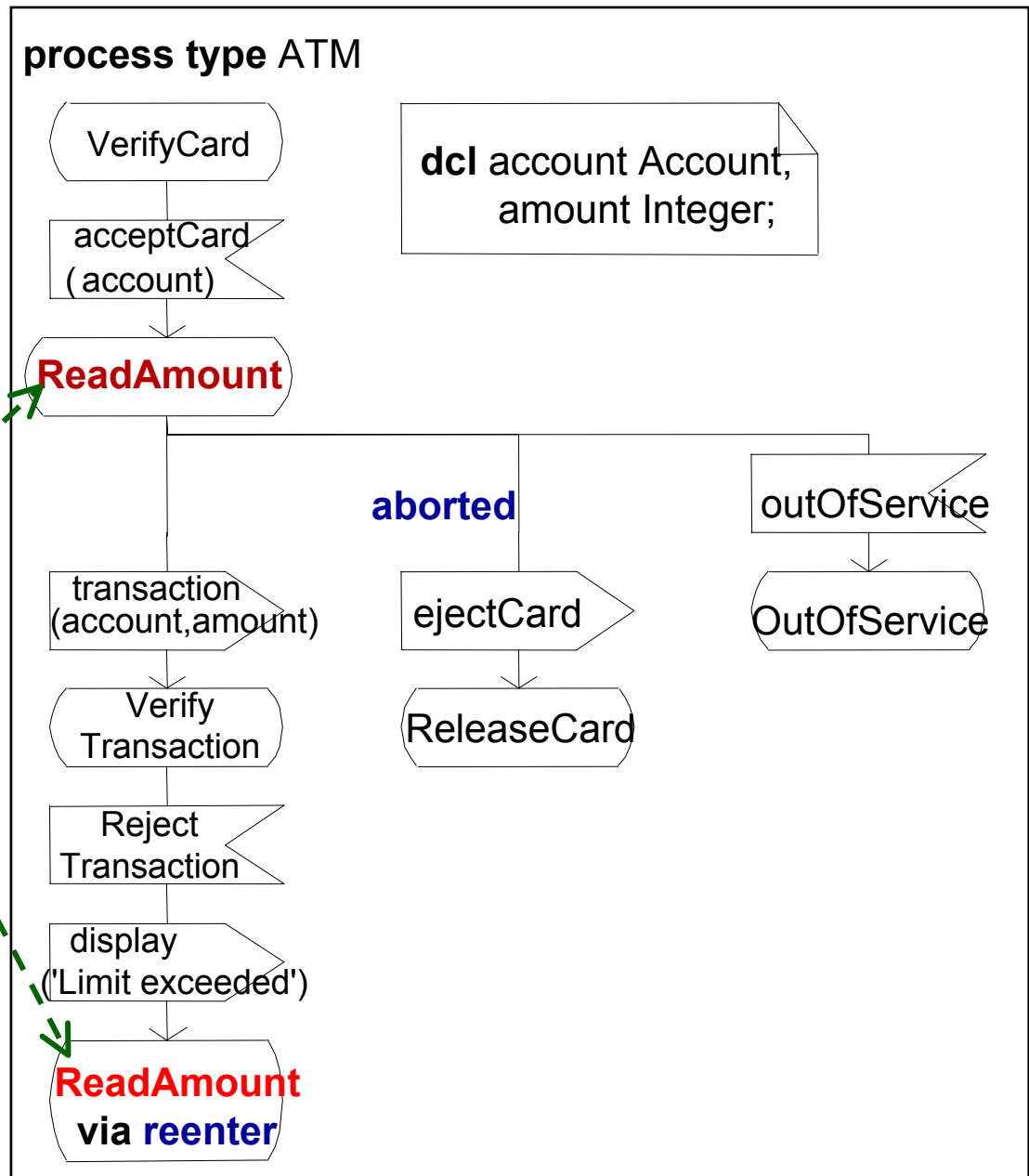
State chart



SDL

Composite States

by means of state references

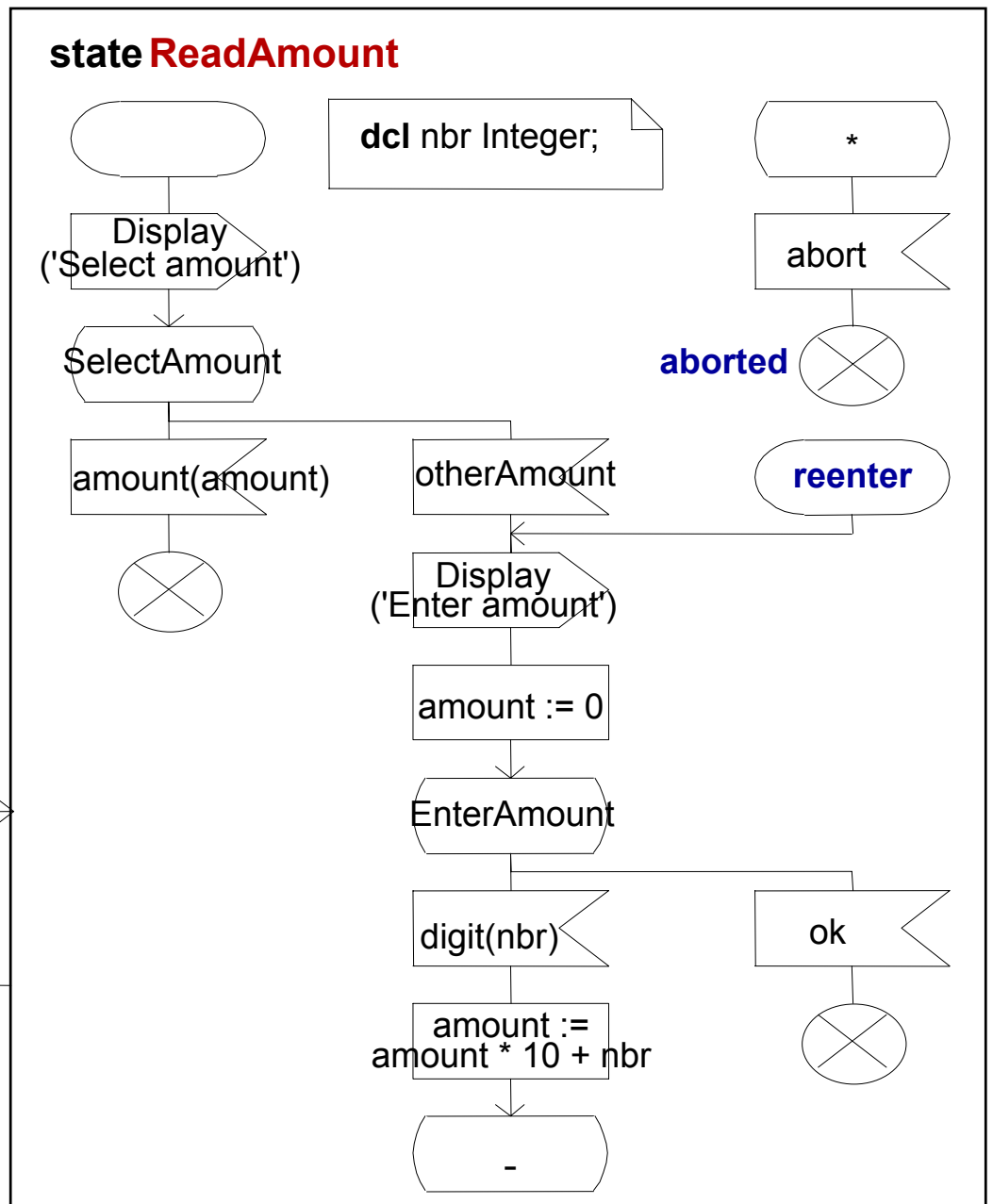


and

Separate State Diagrams

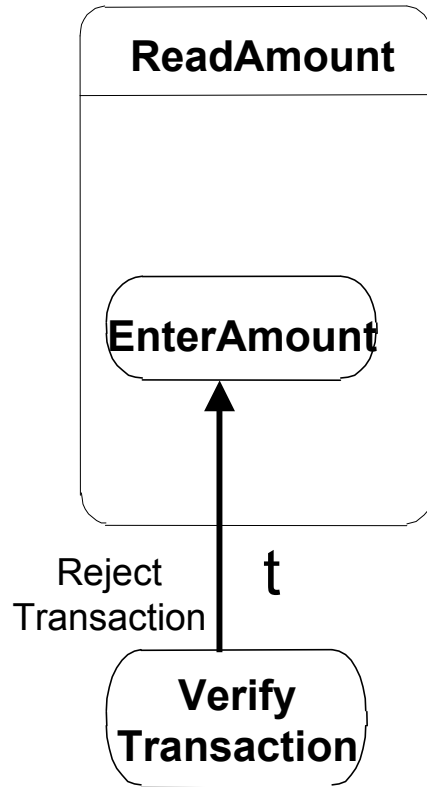
with entry/exit points

- scalability
- encapsulation

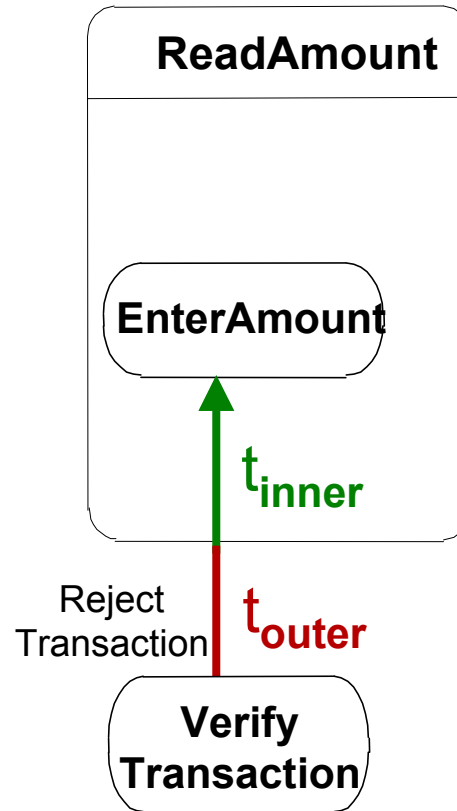


State-boundary-crossing transitions

UML-like

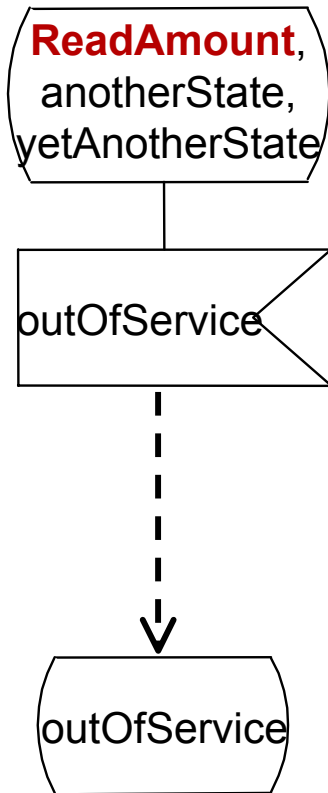


SDL-like

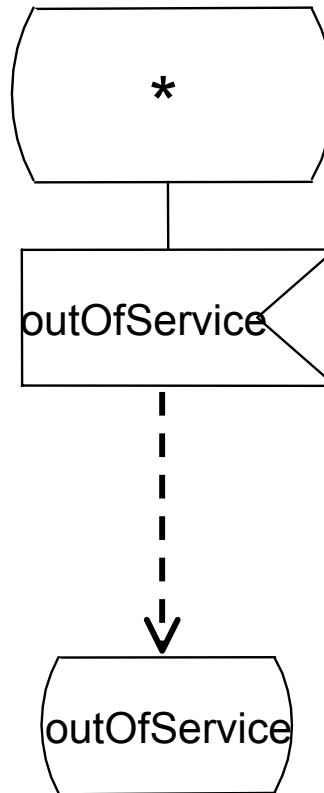


Combines with existing mechanisms

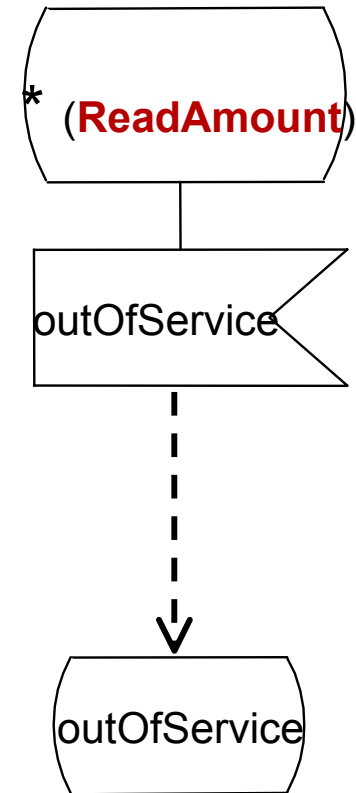
State list



All states



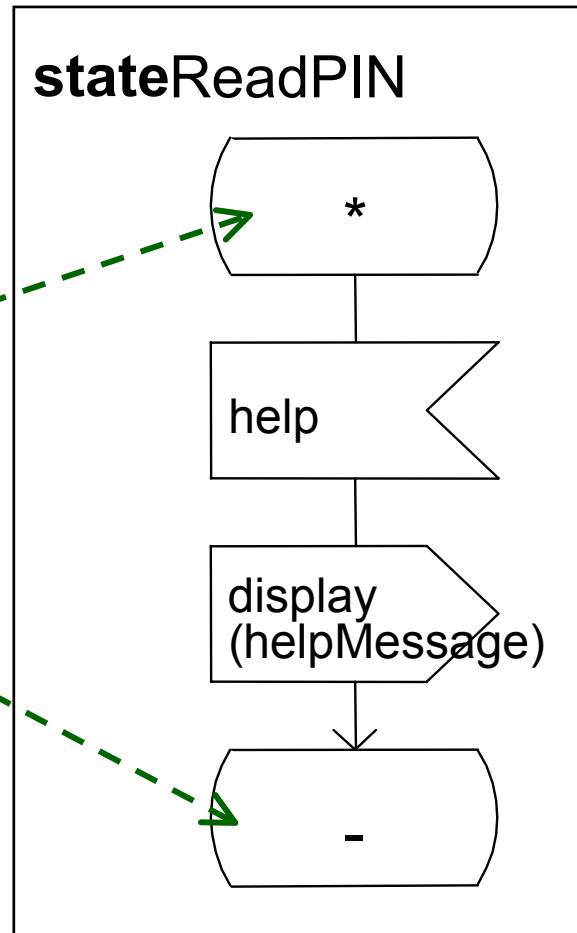
All states, except



Internal transitions

using
existing
mechanisms
of SDL:

- * state
- - nextstate

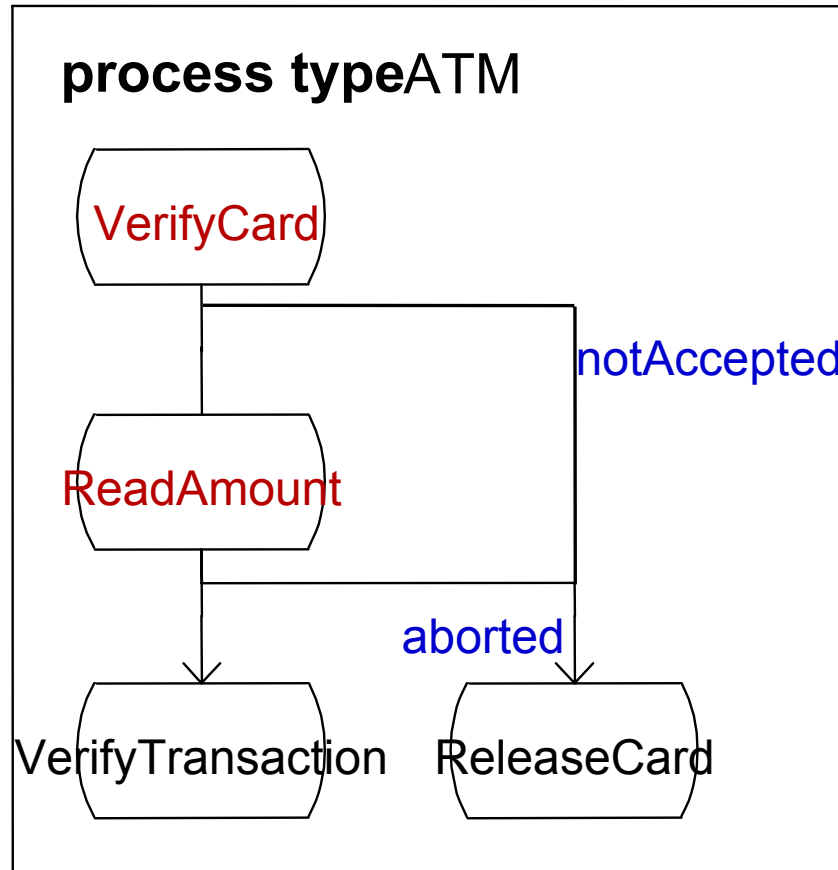


State Overview Diagrams

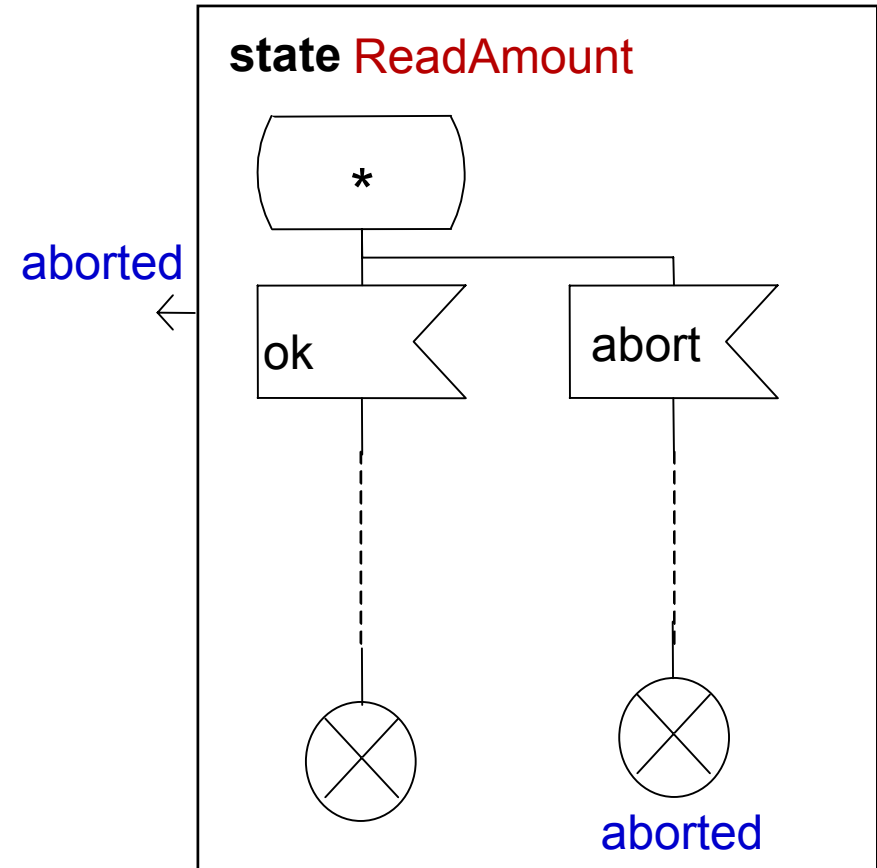
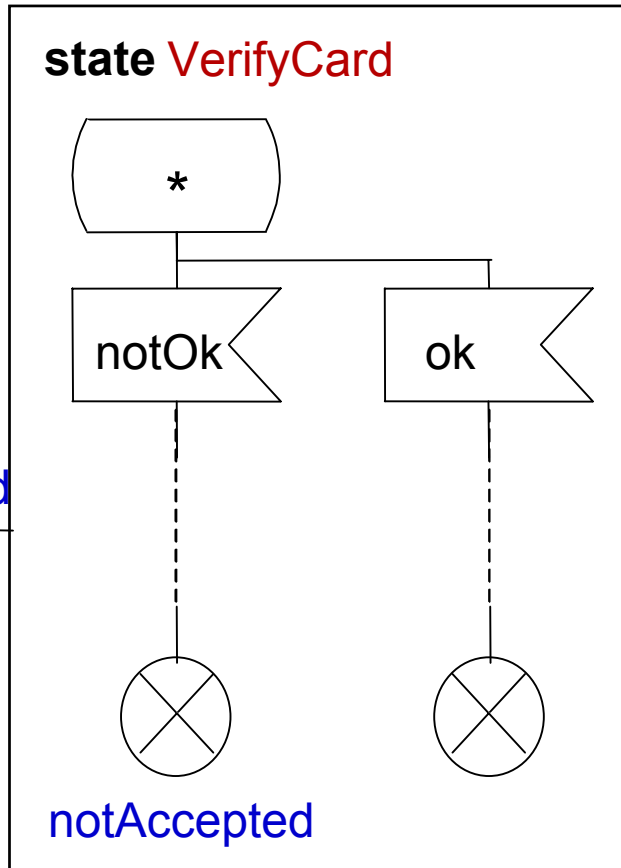
Including only

- states,
- exits from states
- next states

- details of transitions in separate diagrams

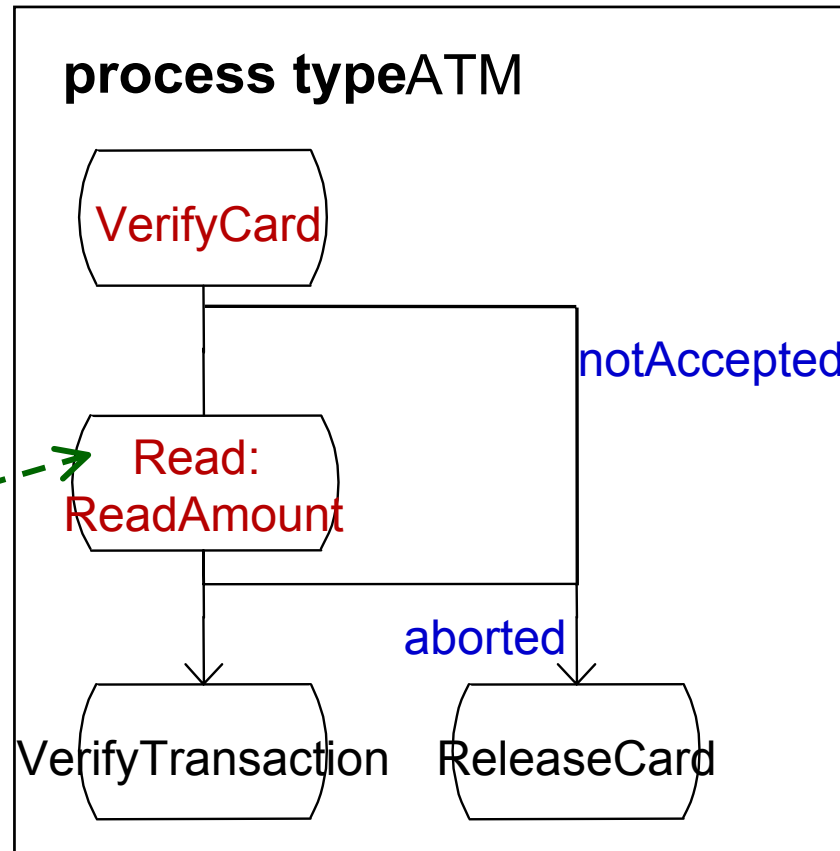


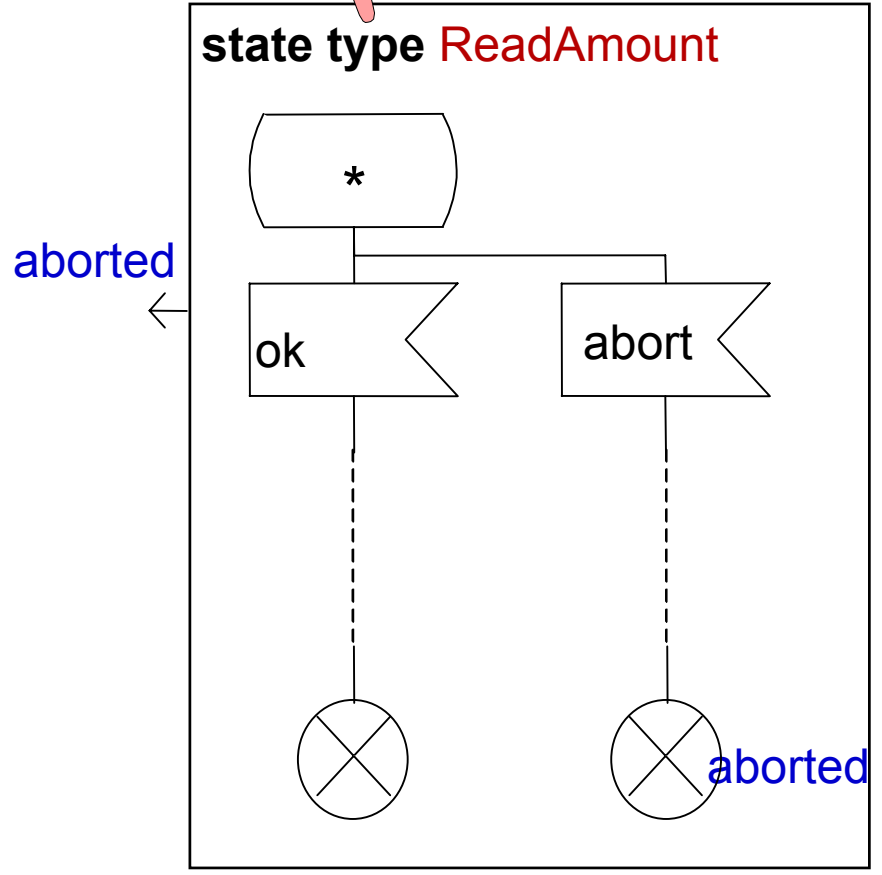
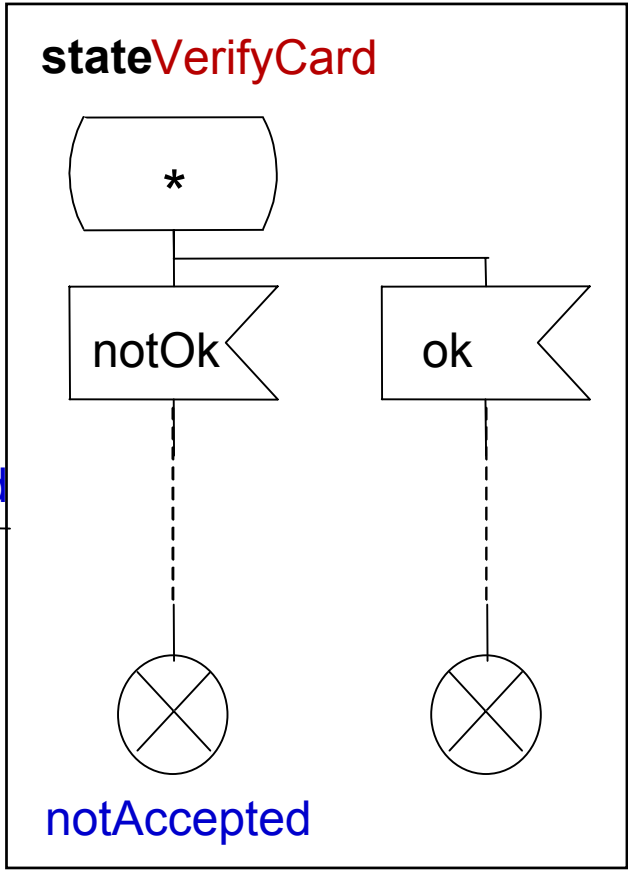
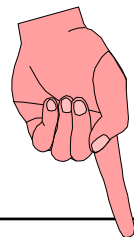
Detailed Transitions in State Diagrams



State types

in order to use the same composite state definition in several situations. States can be type-based.

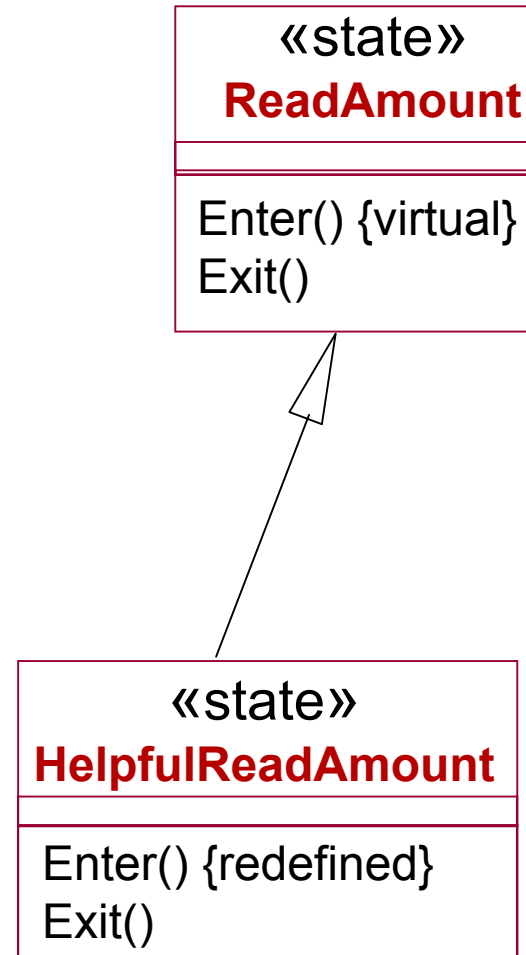




and state type inheritance

for any composite state type, not only the topmost state

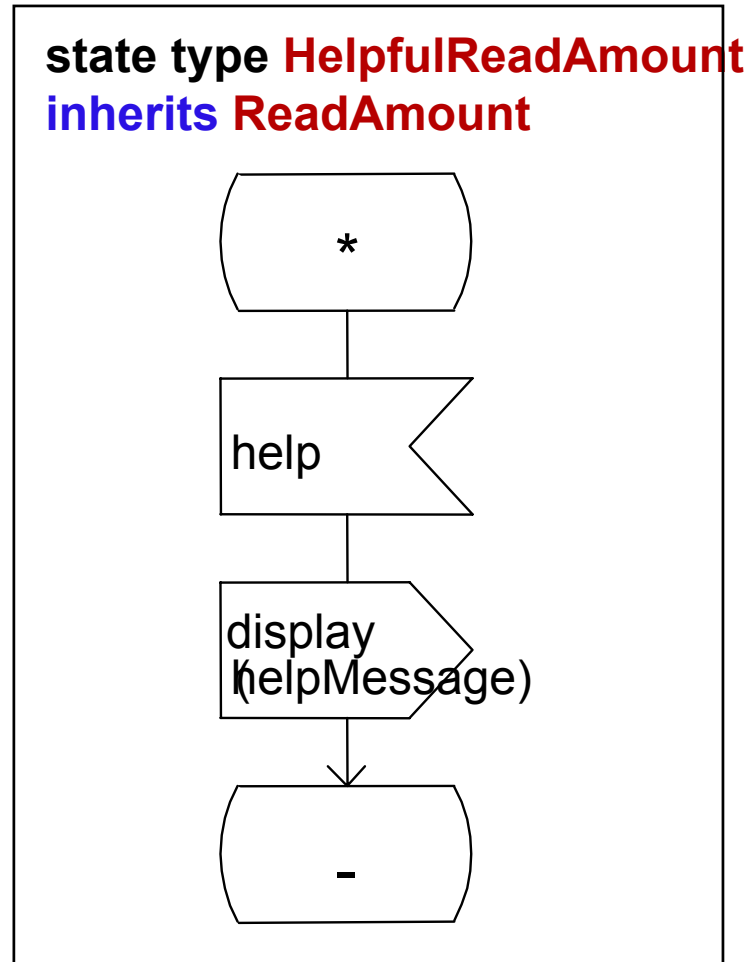
- State type may be a specialisation of supertype by:
 - *inheriting* states and transitions
 - *adding* states and transitions
 - *redefining* virtual procedures, among them enter & exit
 - *redefining* virtual states and virtual transitions



State type inheritance

As it is specified in the state type diagram

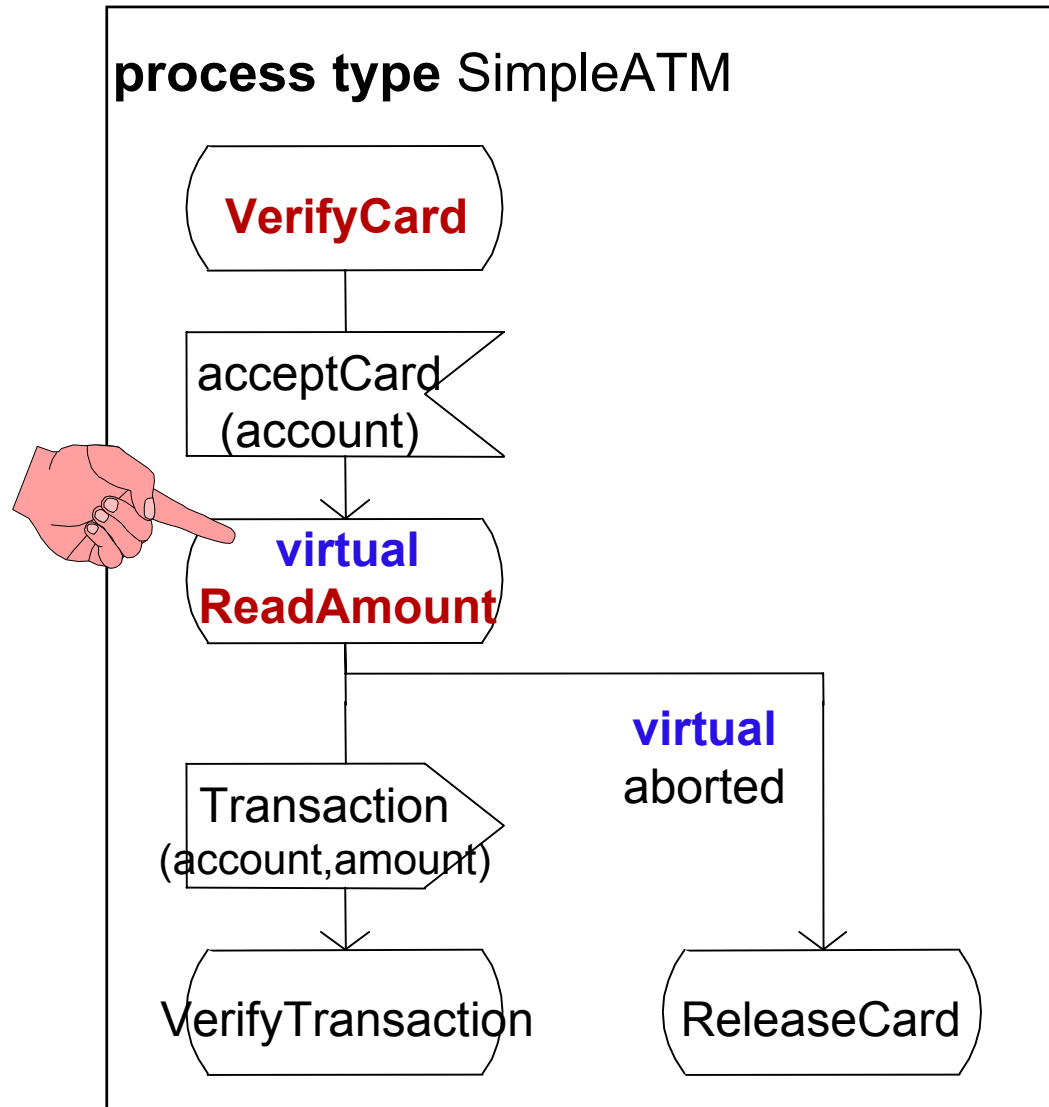
- in this case adding a help transition to all states within HelpfulReadAmount



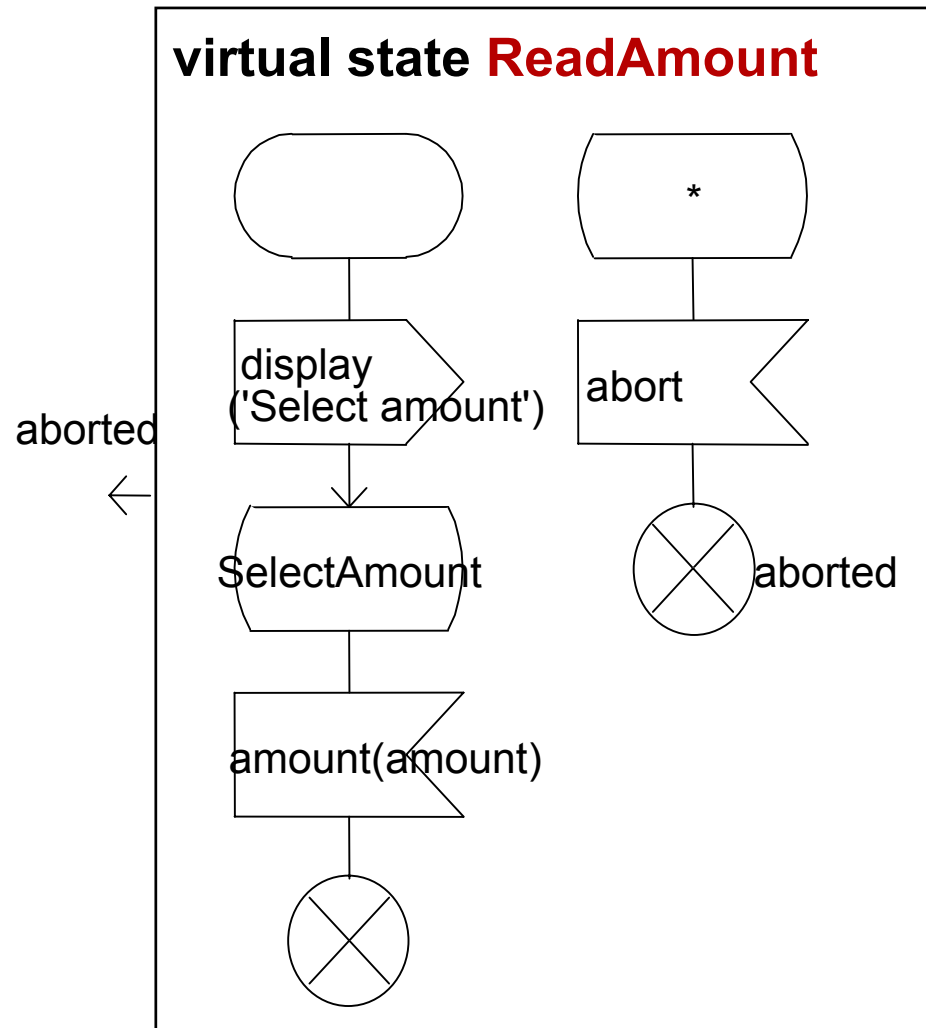
Virtual states

In order to specify which states can be redefined in a specialization.

A virtual state has a constraint, and can only be redefined as an extension of the constraint.



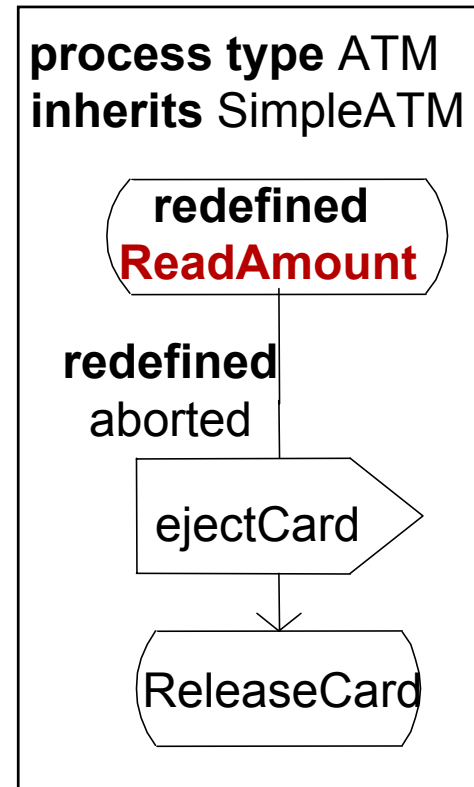
The default constraint is the composite state itself, so redefinitions give extensions of the composite state.



Redefinition of a virtual state

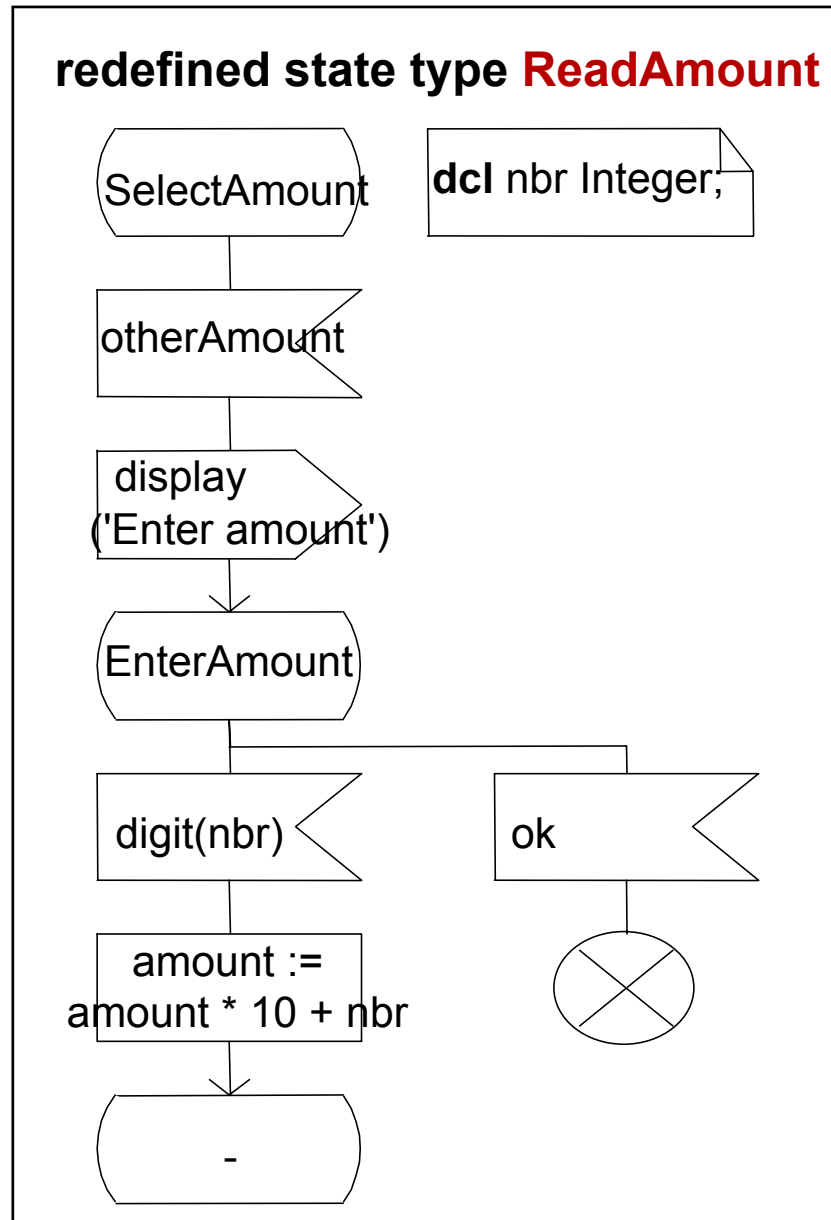
A redefined state is still virtual and can be redefined in a specialization.

Alternatively, a finalized state is a redefinition that can not be further redefined (like Java's final).



State diagram of a redefined state

These states and transitions are added to the states and transitions of the virtual state.



Object oriented data model

- object types - reference assignment
 - value types - value assignment
 - (virtual/redefined/finalised) operators and methods
 - inheritance
-
- Makes SDL independent of implementation language for data handling
 - Provides data modeling in the spirit of Java, C++, with UML-like graphics, SDL-like textual, - and with SDL semantics

Example - object types

```
object type Event struct
    day Day;
    month Month;
    methods
        virtual possible -> Boolean;
endobject type;
```

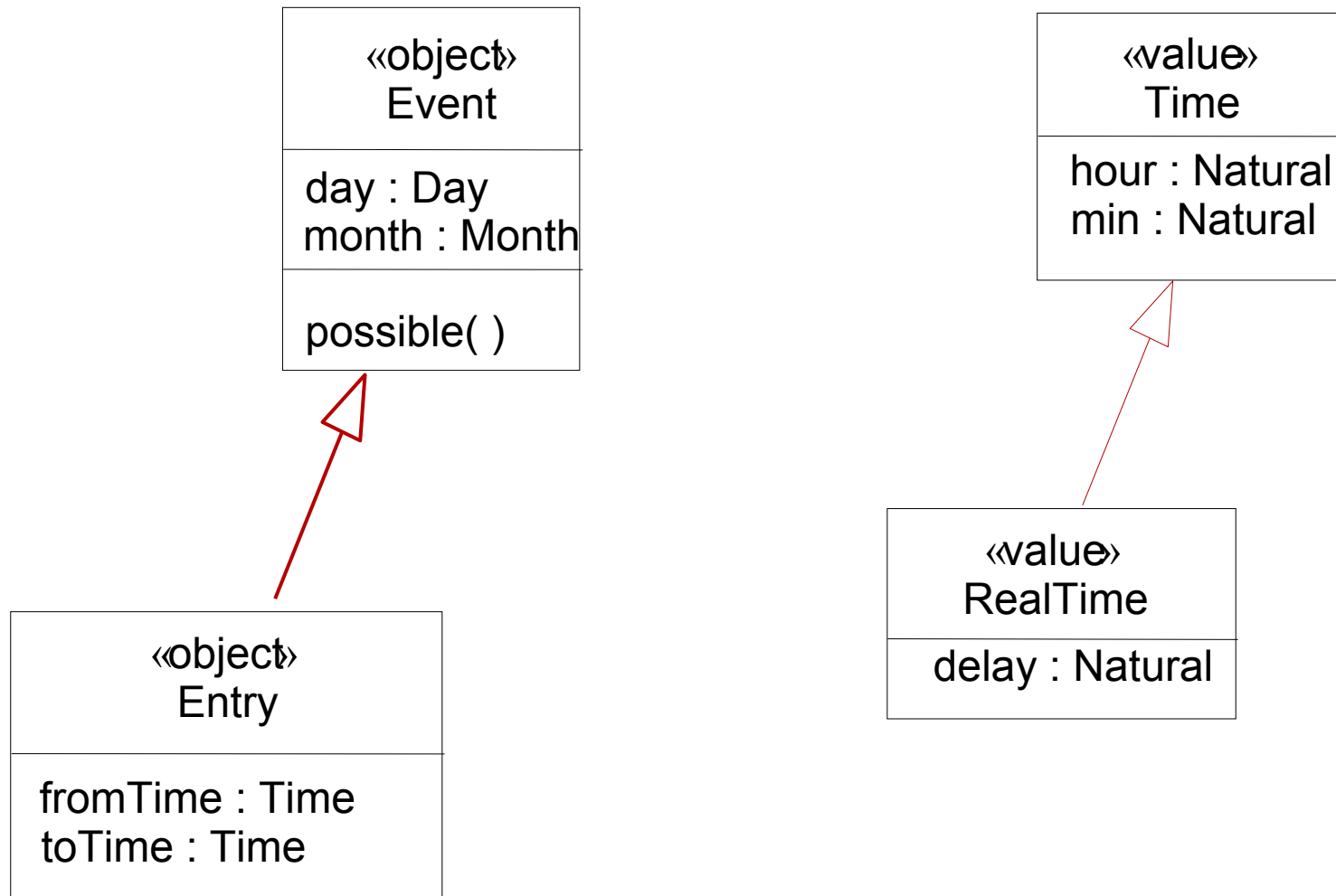
```
object type Appointment inherits Event
    fromTime, toTime Time;
    methods
        redefined possible;
endobject type;
```

Example - value types

```
value type Time struct
    hour Natural;
    minute Natural;
endvalue type;
```

```
value type RealTime inherits Time
    delay Natural;
endvalue type;
```

Data types – also by means of class symbols



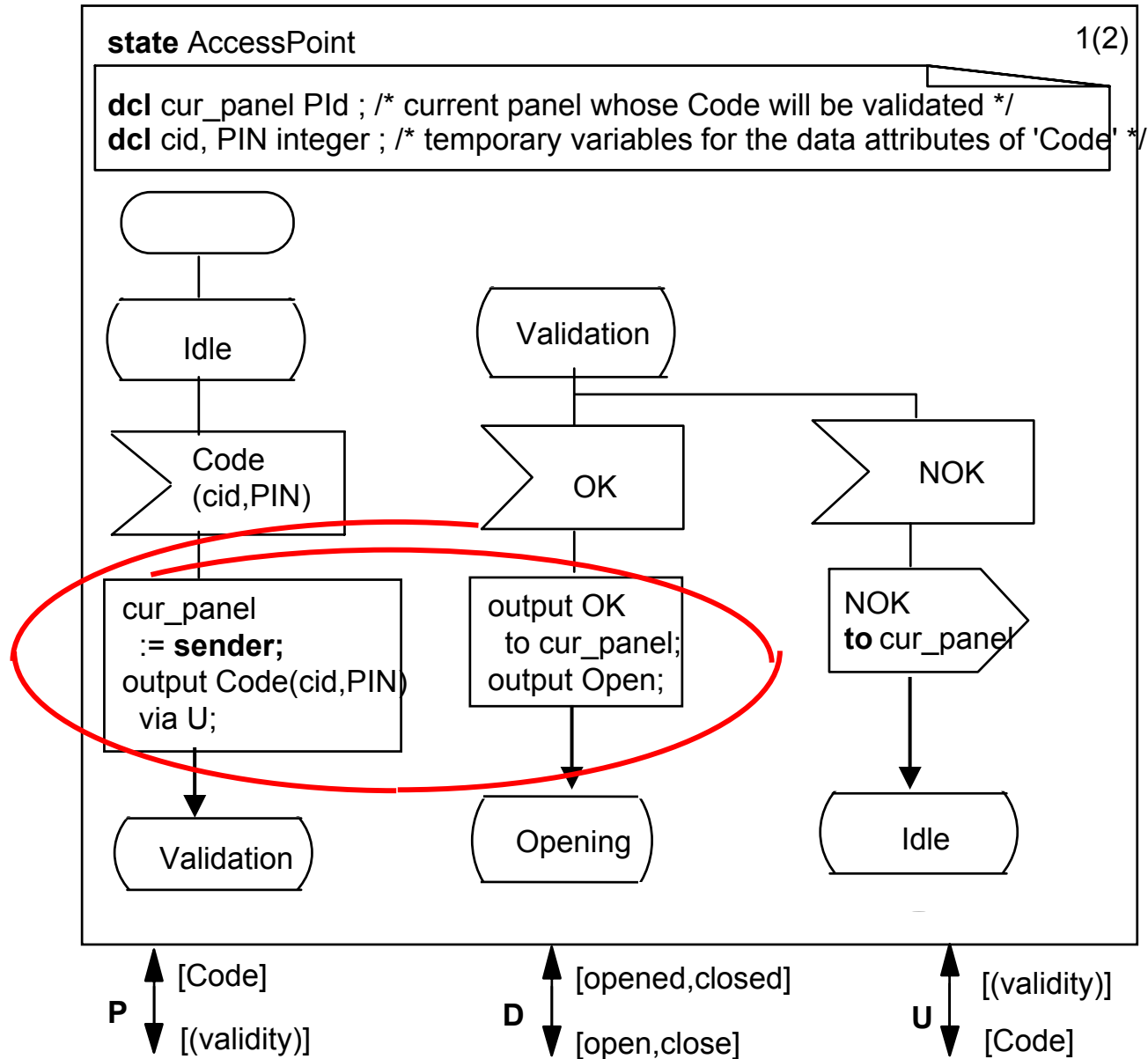
Action language I

- Action language (textual) in task symbols and procedures
- C++, Java like syntax, compounds, loop constructs, etc.

```
decl i1, i2 Integer;  
  i1 := (x / 5) + 1;  
  i1 := i1 * 2;  
  ...  
  { ... }  
...
```

- Makes SDL independent of C and any implementation language (a closed language)

Action language II



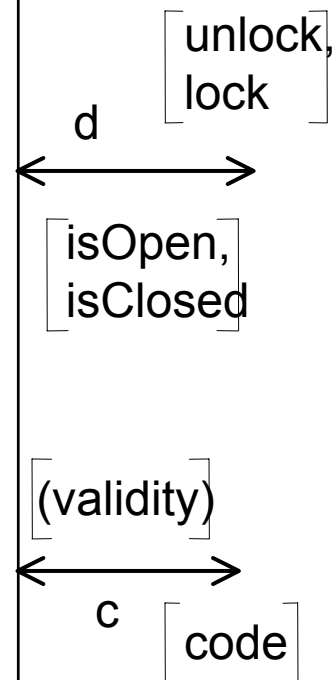
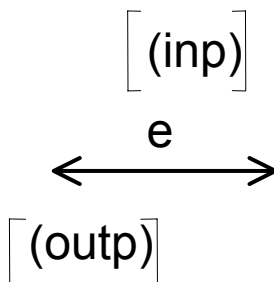
Agents

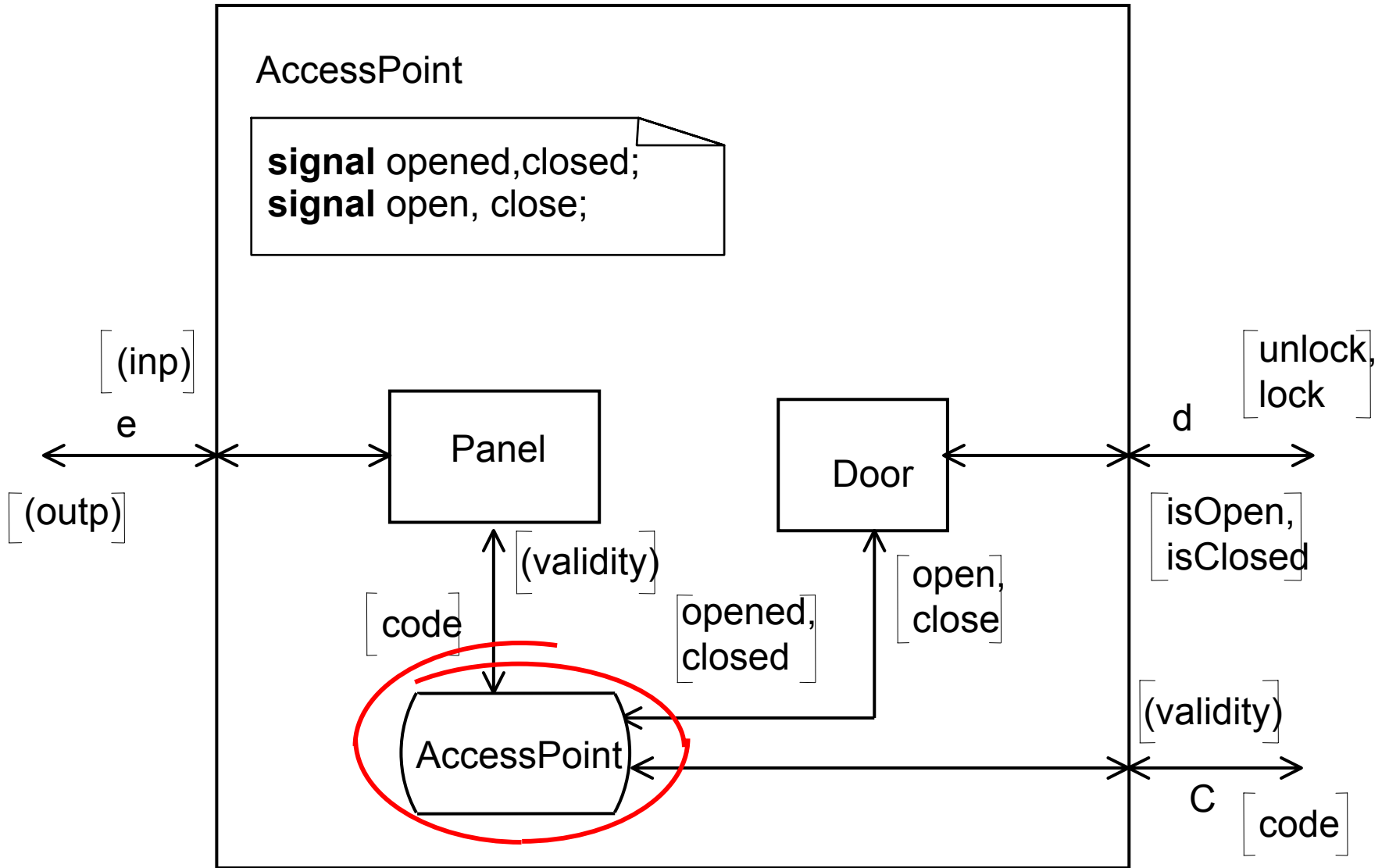
AccessPoint

Agents:

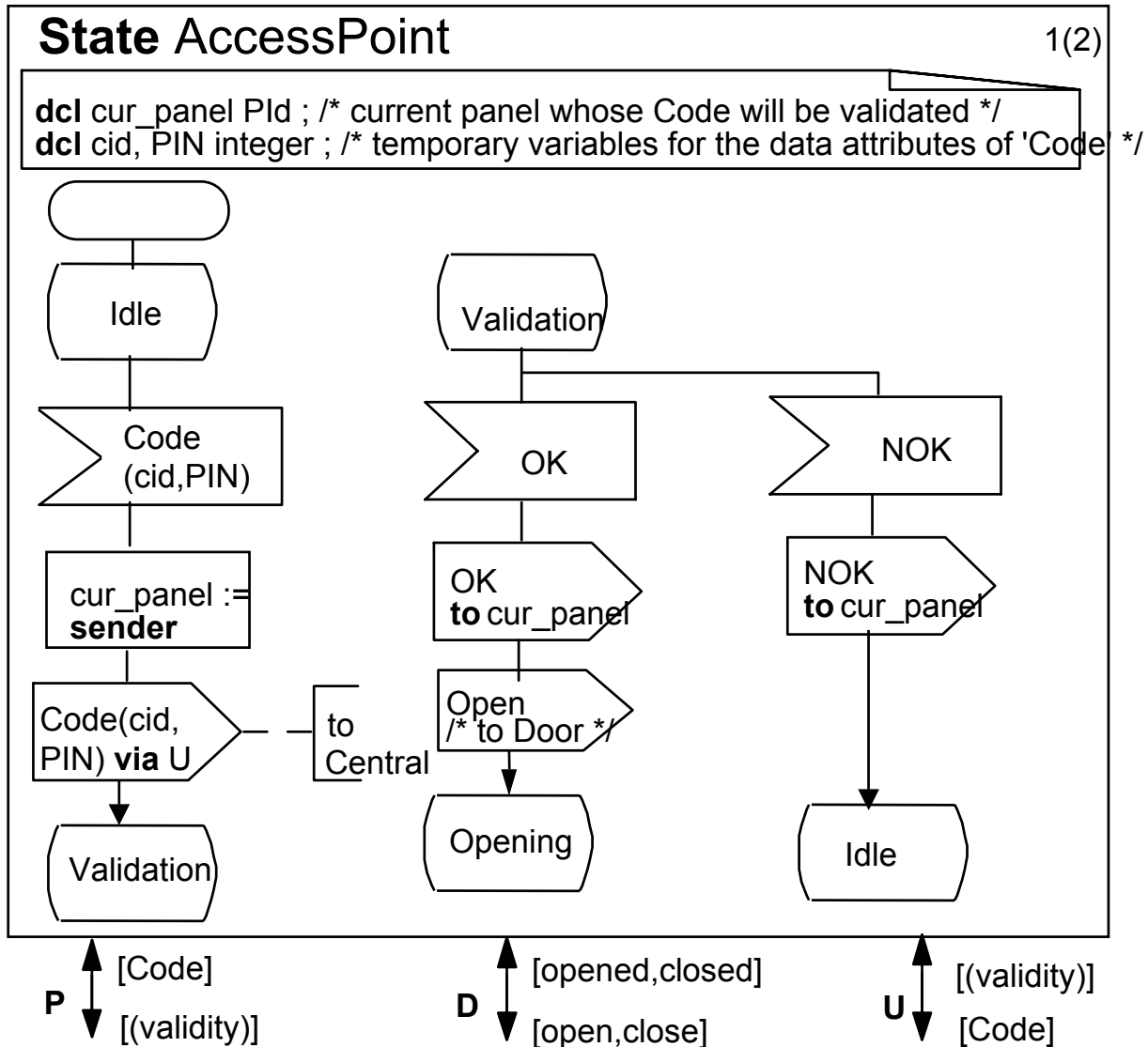
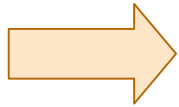
- the main objects of SDL-2000
- unifies system, block, process, service
- has either behaviour
- or agent structure
- or both

Specified from the outside by means of interfaces and gates

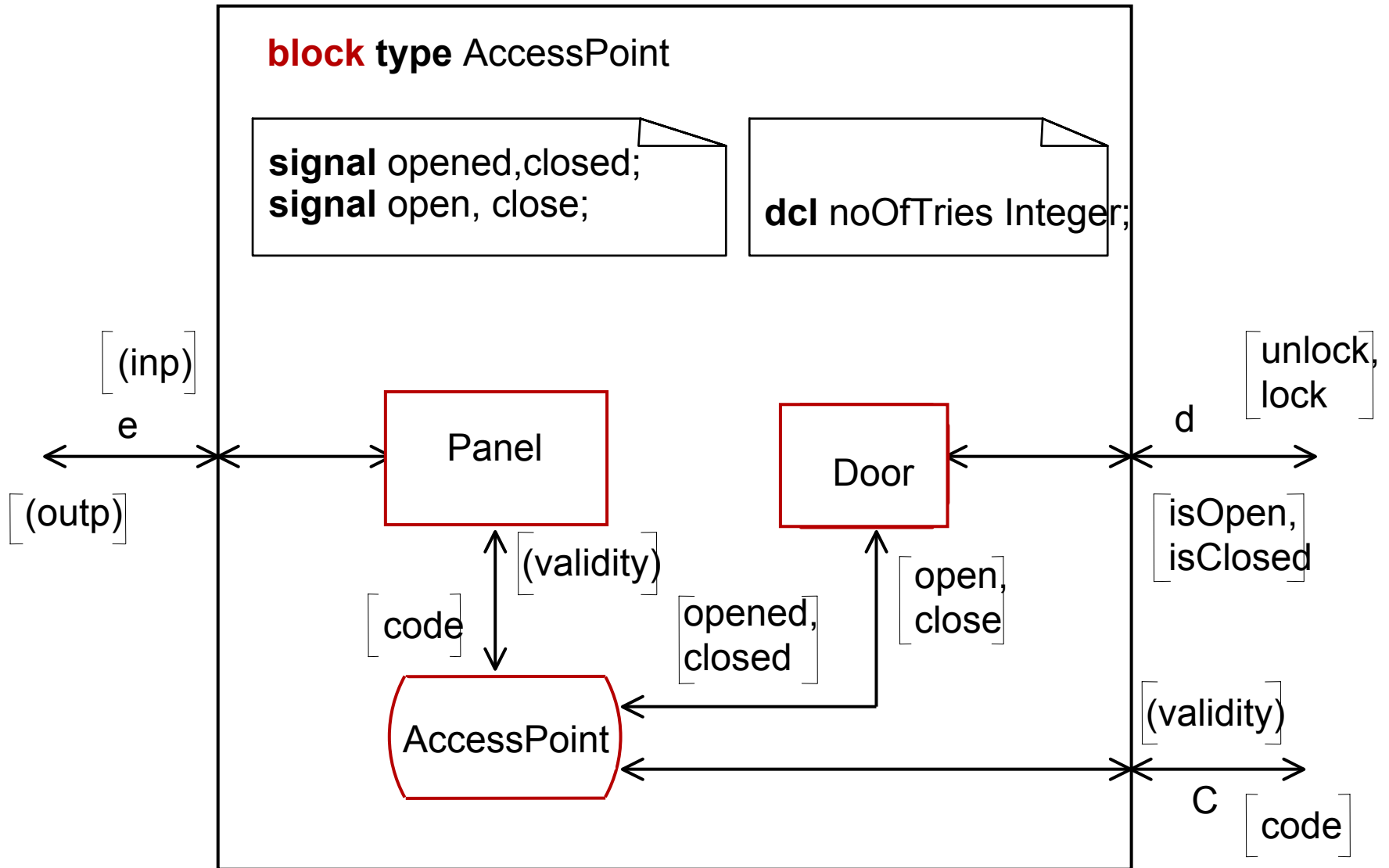




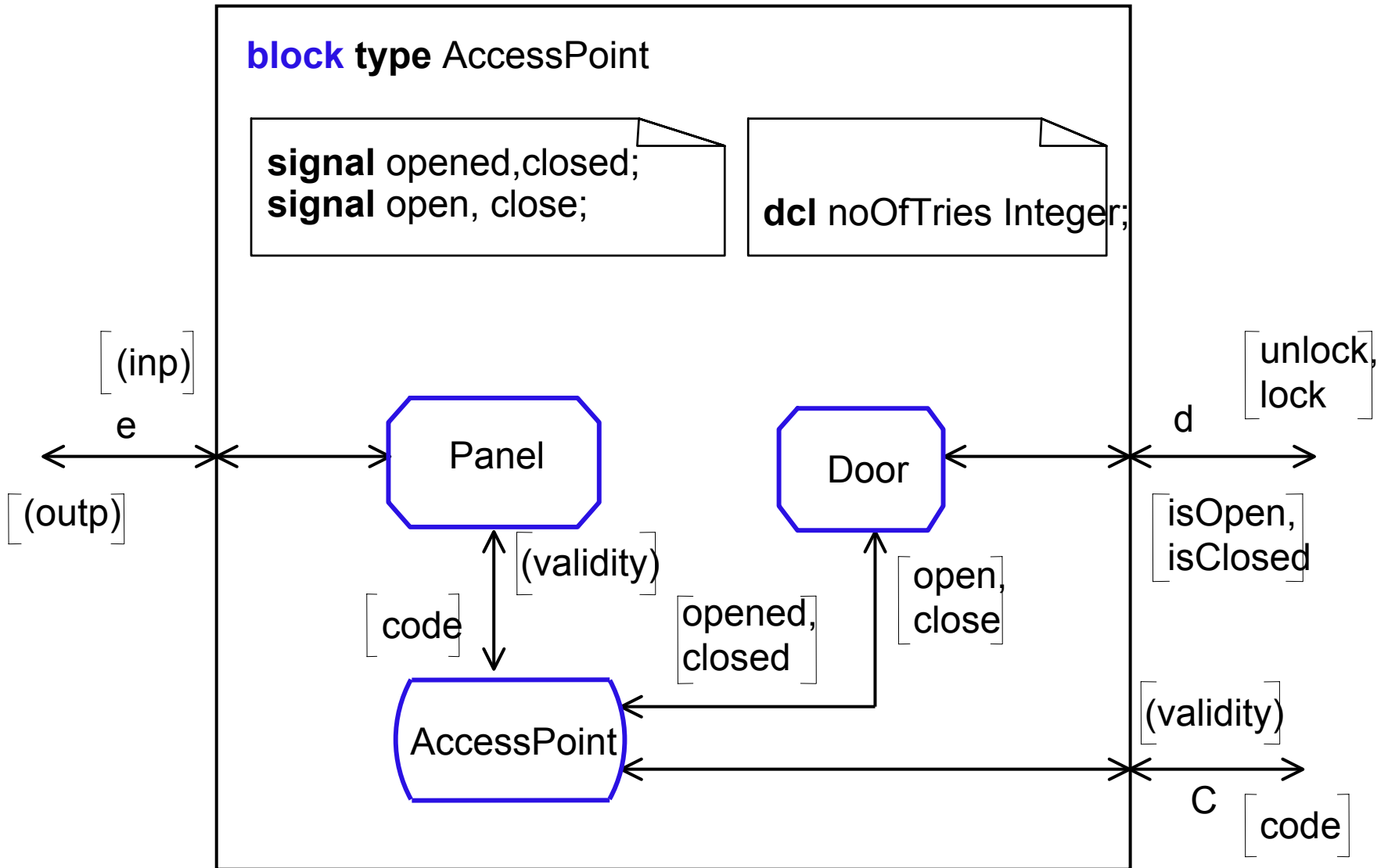
Actor behaviour as state



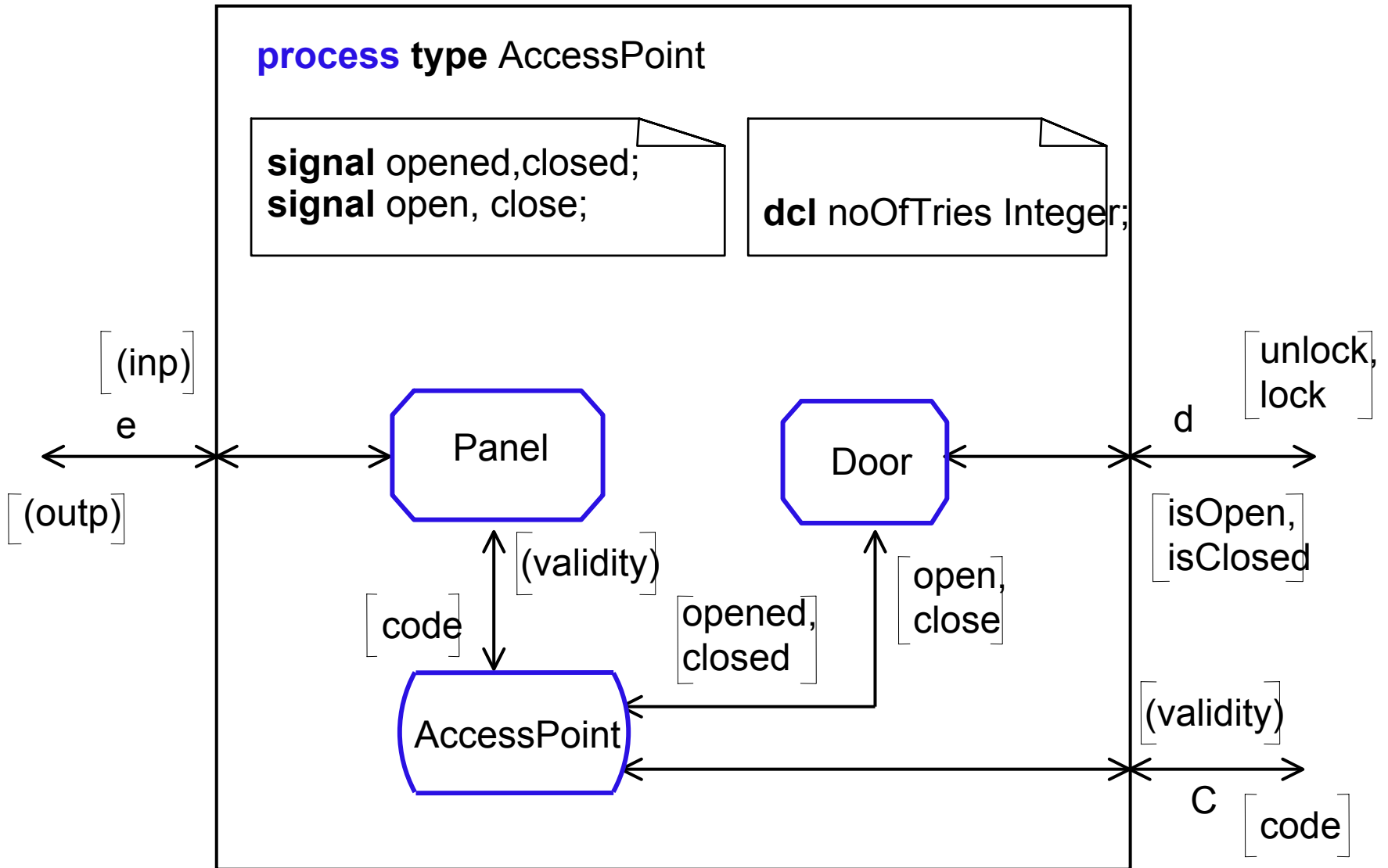
Concurrent or alternating entities I



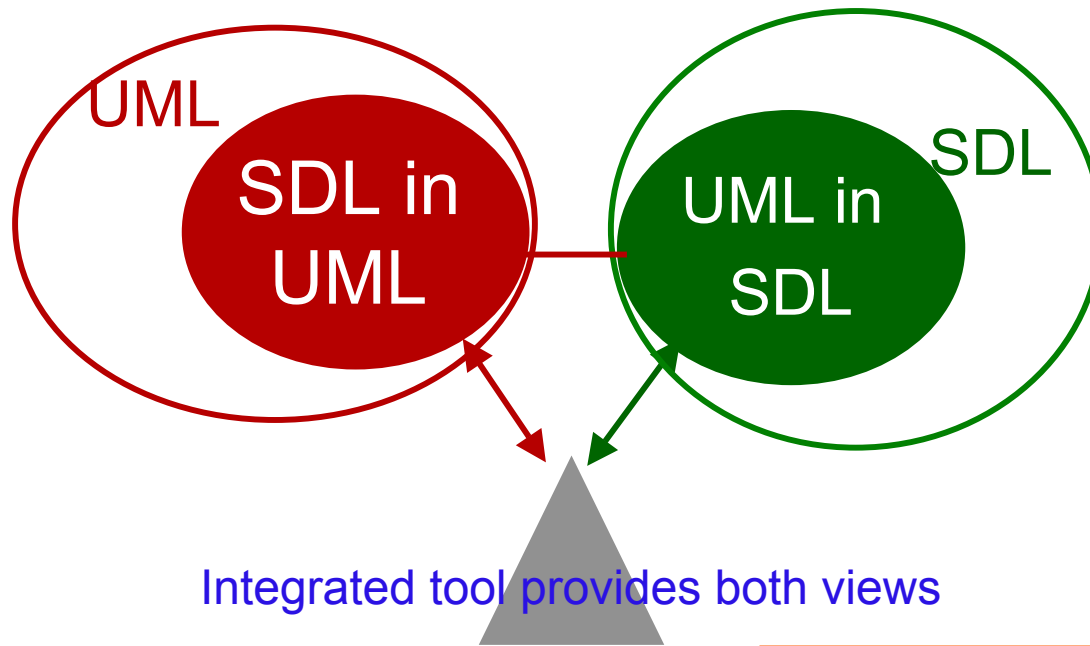
Concurrent or alternating entities II



Concurrent or alternating entities III



SDL combined with UML



UML for:

- Classes (type references) and Associations

SDL for:

- Systems
- Composite types
- Object structures
- Complete behaviour

MSC for:

- Message sequences

How this is achieved

- **SDL UML Profile (Z.109)**
 - *Specialisation of UML that maps to SDL*
 - *by stereotypes: «system», «block», «process», «service», «procedure», «signal», «object», «value», «state»*
- **SDL Graphics Extensions (Z.100)**
 - *Class symbols as combined type references and partial type definitions*
 - *Package symbols and dependency between these*
 - *Generalisation relationship for specialisation*
 - *Parameterised class for type with context parameters*
 - *Composition for containment*
 - *General associations as comments*
- **Approved by ITU-T SG10 November 1999**