# 8 Specifying with natural language

The system shall ...

The oldest...

...and most widely used way

- taught at school
- extremely expressive

But not necessarily the best

- Ambiguous
- Imprecise
- Error-prone
- Verification primarily by careful reading

Michelangelo's Moses (San Pietro in Vincoli, Rome)
Moses holds the Ten Commandments in his hand: written in natural language

# Problems with natural language requirements

Read the subsequent requirements. Any findings?

"For every turnstile, the total number of turns shall be read and archived once per day."

"The system shall produce lift usage statistics."

"Never shall an unauthorized skier pass a turnstile."

"By using RFID technology, ticket validation shall become faster."

"In the sales transaction, the system shall record the buyer's data and timestamp the sold access card."

# Some rules for specifying in natural language

[Rupp et al. 2009]

- Use active voice and defined subjects
- Build phrases with complete verbal structure
- Use terms as defined in the glossary
- Define precise meanings for auxiliary verbs (shall, should, must, may,...) as well as for process verbs (for example, “produce”, “generate”, “create”)
- Check for nouns with unspecific semantics (“the data”, “the customer”, “the display”,...) and replace where appropriate
- When using adjectives in comparative form, specify a reference point: “better” ➜ “better than”

# More rules

- Scrutinize all-quantifications: “every”, “always”, “never”, etc. seldom hold without any exceptions
- Scrutinize nominalizations (“authentication”, “termination”...): they may conceal incomplete process specifications
- State every requirement in a main clause. Use subordinate clauses only for making the requirement more precise
- Attach a unique identifier to every requirement
- Structure natural language requirements by ordering them in sections and sub-sections
- Avoid redundancy where possible: “never ever” ➜ “never”

# Phrase templates

[Rupp et al. 2009
ISO/IEC/IEEE 29148:2011]

Use templates for creating well-formed natural language requirements

Typical template:

[<Condition>] <Subject> <Action> <Objects> [<Restriction>]

Example:

When a valid card is sensed, the system shall send the command ‘unlock_for_a_single_turn’ to the turnstile within 100 ms.

# Agile stories

[Cohn 2004]

- A single sentence about a requirement
- Written from a stakeholder's perspective
- Optionally including the expected benefit
- Accompanied by acceptance criteria for requirement
- Acceptance criteria make the story more precise

Standard template:

As a <role> I want to <my requirement> [ so that <benefit> ]

# A sample story

As a skier, I want to pass the chairlift gate so that I get access without presenting, scanning or inserting a ticket at the gate.

Author: Dan Downhill Date: 2013-09-20 ID: S-18

# Sample acceptance criteria

Acceptance criteria:

- Recognizes cards worn anywhere in a pocket on the left side of the body in the range of 50 cm to 150 cm above ground
- If card is valid: unlocks turnstile and flashes a green light for five seconds or until the turnstile is moved
- If card is invalid: doesn't unlock gate and flashes a red light for five seconds
- Time from card entering the sensor range until unlock and flash red or green is less than 1.5 s (avg) & 3 s (max)
- The same card is not accepted twice within an interval of 200 s

# Mini-Exercise: Writing a user story

Consider the chairlift access control case study.

Write a story from a skier's perspective about buying a day card.

# All-quantification and exclusion

- Specifications in natural language frequently use all-quantifying or excluding statements without much reflection:

  "When operating the coffee vending machine, the user shall always be able to terminate the running transaction by pressing the cancel key."

- ➪ Scrutinize all-quantifications ("every", "all", "always"...) and exclusions ("never", "nobody", "either – or",...) for potential exceptions
- ➪ Specify found exceptions as requirements

# Dealing with redundancy

- Natural language is frequently (and deliberately) redundant

  → Secures communication success in case of some information loss

- In requirements specifications, redundancy is a problem
  - Requirements are specified more than once
  - In case of modifications, all redundant information must be changed consistently

- Make redundant statements only when needed for abstraction purposes

- Avoid local redundancy: “never ever” → “never”

# 9 Model-based requirements specification

A guided tour through ...

- Data and object modeling
- Behavior modeling
- Function and process modeling
- User interaction modeling
- Goal modeling
- UML

Primarily for functional requirements

Quality requirements and constraints are mostly specified in natural language

# 9.1 Characteristics and options

- Requirements are described as a problem-oriented model of the system to be built
- Architecture and design information is omitted
- Mostly graphically represented
- Semi-formal or formal representation

# What can be modeled?

System view: modeling a system's static structure, behavior and functions

**Static structure perspective**
- (Entity-Relationship) data models
- Class and object models
- Sometimes component models

**Behavior perspective**
- Finite state machines
- Statecharts / state machines
- Petri nets

**Function and flow perspective**
- Activity models
- Data flow / information flow models
- Process and work flow models

# What can be modeled? – continued

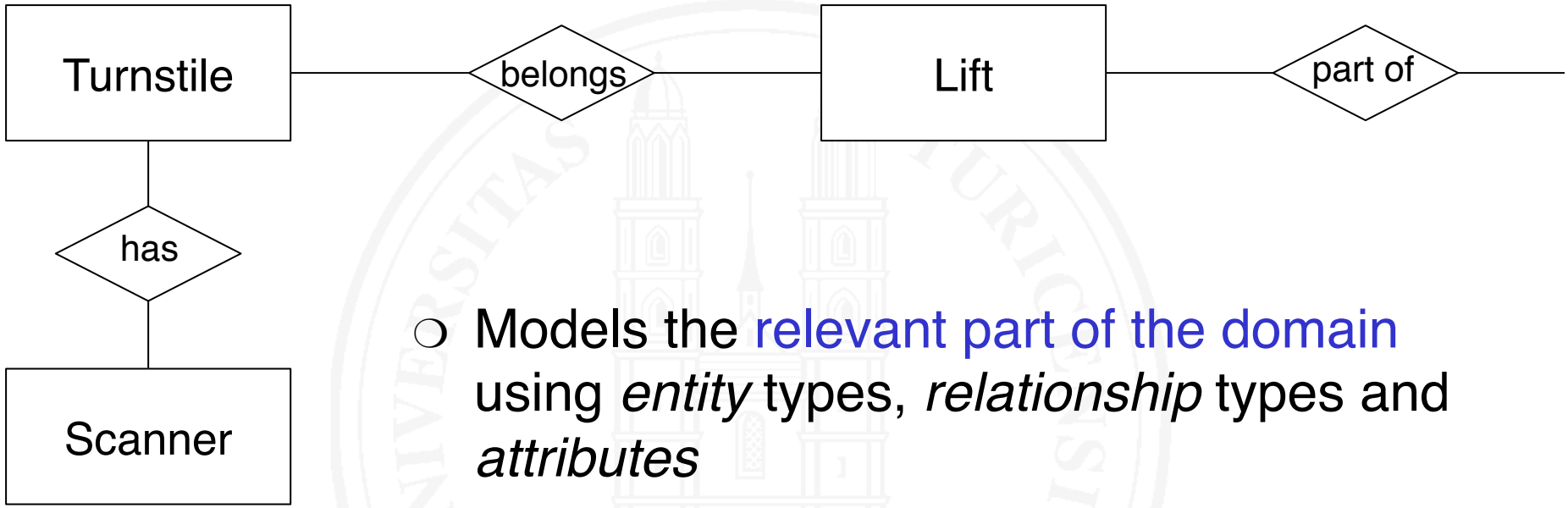
- User-system interaction view: modeling the interaction between a system and its external actors
  - Use cases, scenarios
  - Sequence diagrams
  - Context models
- Goal view: modeling goals and their dependencies
  - Goal trees
  - Goal-agent networks, e.g., i*

# 9.2 Models of static system structure

- Entity-relationship models
- Class and object models
- Component models

# Data modeling (entity-relationship models)

[Chen 1976]




- ❍ Models the relevant part of the domain using *entity* types, *relationship* types and *attributes*
- \+ Rather easy to model
- \+ Straightforward mapping to relational database systems
- – Ignores functionality and behavior
- – No means for system decomposition

# Object and class modeling

[Booch 1986, Booch 1994, Glinz et al. 2002]

Idea

- Identify those entities in the domain that the system has to store and process
- Map this information to objects/classes, attributes, relationships and operations
- Represent requirements in a static structural model
- Modeling individual objects does not work: too specific or unknown at time of specification
  - → *Classify* objects of the same kind to classes: Class models
  - → or select an abstract *representative*: Object models

# Terminology

**Object** – an individual entity which has an identity and does not depend on another entity.

Examples: Turnstile no. 00231, The Plauna chairlift

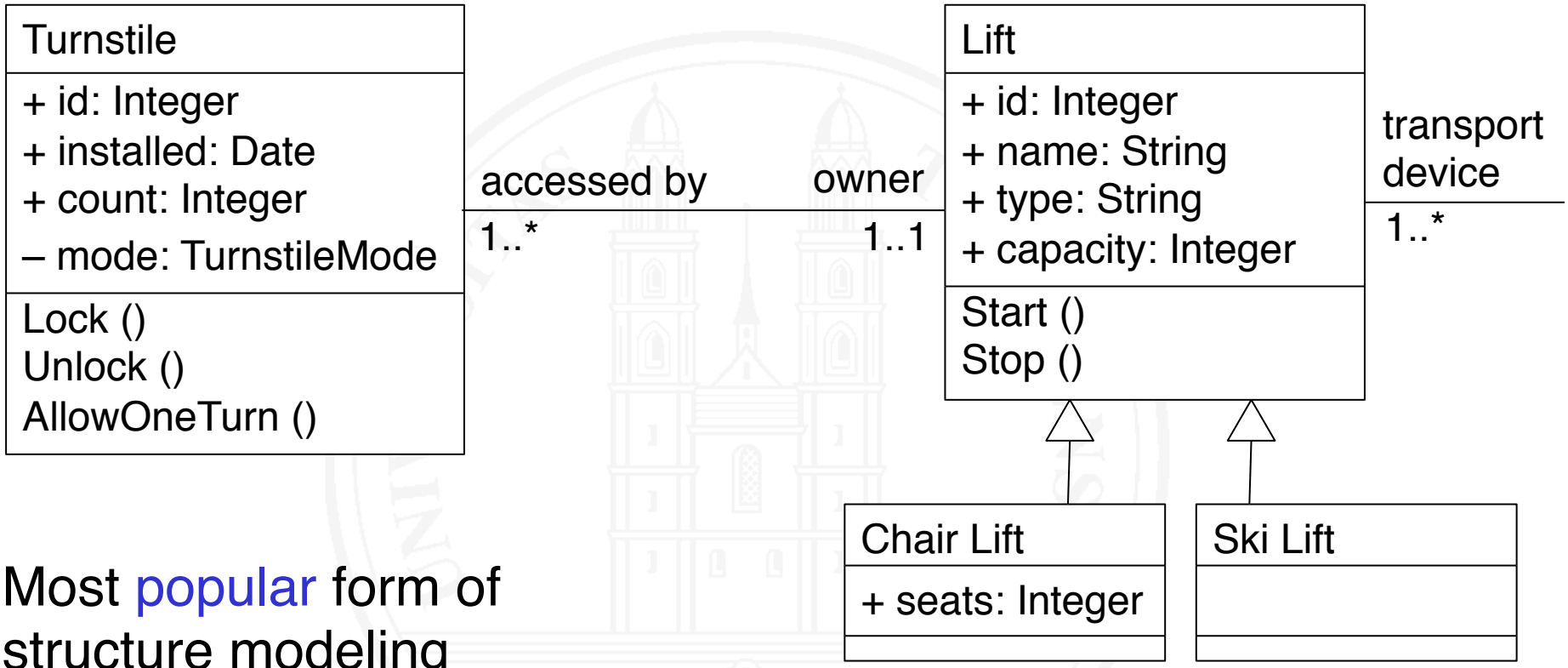
**Class** – Represents a set of objects of the same kind by describing the structure of the objects, the ways they can be manipulated and how they behave.

Examples: Turnstile, Lift

**Abstract Object** – an abstract representation of an individual object or of a set of objects having the same type

Example:  A Turnstile

# Class models / diagrams




Most popular form of structure modeling

Typically using UML class diagrams

Class diagram: a diagrammatic representation of a class model

# Class models are sometimes inadequate

- Class models don't work when different objects of the same class need to be distinguished
- Class models can't be decomposed properly: different objects of the same class may belong to different subsystems
- Subclassing is a workaround, but no proper solution

In such situations, we need object models

# Object models: a motivating example

Example: Treating incidents in an emergency command and control system

Emergency command and control systems manage incoming emergency calls and support human dispatchers in reacting to incidents (e.g., by sending police, fire fighters or ambulances) and monitoring action progress.

When specifying such a system, we need to model

- Incoming incidents awaiting treatment
- The incident currently managed by the dispatcher
- Incidents currently under treatment
- Closed incidents

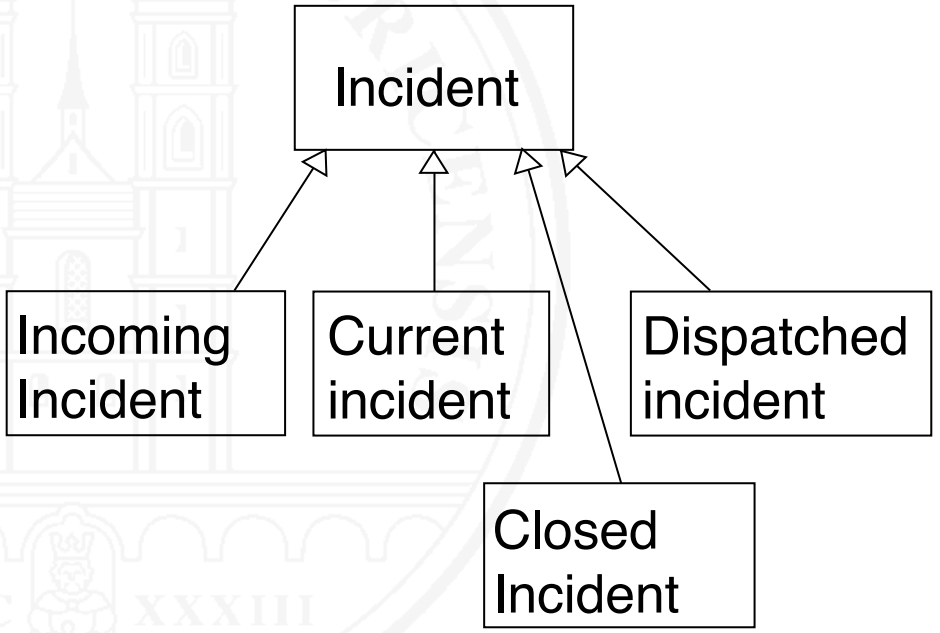
# Class models are inadequate here

In a class model, incidents would have to be modeled as follows:

either


Bad: essential elements of the problem are not modeled

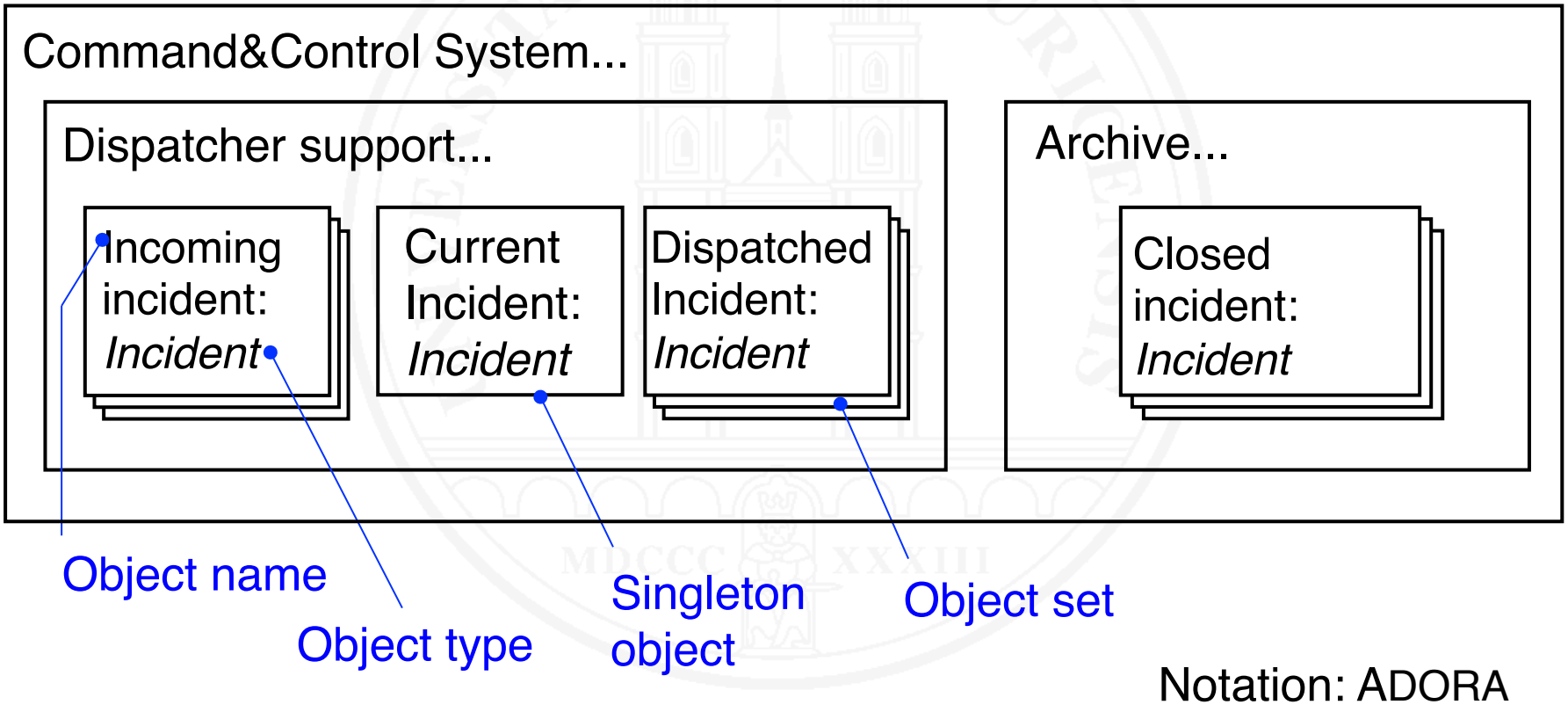
or




Unnatural: all subclasses are structurally identical

# Object models work here

Modeling is based on a hierarchy of abstract objects




Notation: ADORA

# ADORA

[Glinz et al. 2002]

- ADORA is a language and tool for object-oriented specification of software-intensive systems
- Basic concepts
  - Modeling with abstract objects
  - Hierarchic decomposition of models
  - Integration of object, behavior and interaction modeling
  - Model visualization in context with generated views
  - Adaptable degree of formality
- Developed in the RERG research group at UZH

# Modeling with abstract objects in UML

- Not possible in the original UML (version 1.x)
- Introduced 2004 as an option in UML 2
- Abstract objects are modeled as components in UML
- The component diagram is the corresponding diagram
- Lifelines in UML 2 sequence diagrams are also frequently modeled as abstract objects
- In UML 2, class diagrams still dominate

# What can be modeled in class/object models?

- Objects as *classes* or *abstract objects*
- Local properties as *attributes*
- Relationships / non-local properties as *associations*
- Services offered by objects as *operations* on objects or classes (called *features* in UML)
- Object behavior
  - Must be modeled in separate *state machines* in UML
  - Is modeled as an *integral part* of an object hierarchy in ADORA
- System-context interfaces and functionality from a user's perspective *can't* be modeled *adequately*

# Object-oriented modeling: pros and cons

+ Well-suited for describing the structure of a system
+ Supports locality of data and encapsulation of properties
+ Supports structure-preserving implementation
+ System decomposition can be modeled
– Ignores functionality and behavior from a user's perspective
– UML class models don't support decomposition
– UML: Behavior modeling weakly integrated

# Mini-Exercise: Classes vs. abstract objects

Specify a distributed heating control system for an office building consisting of a central boiler control unit and a room control unit in every office and function room.

- The boiler control unit shall have a control panel consisting of a keyboard, a LCD display and on/off buttons.
- The room control unit shall have a control panel consisting of a LCD display and five buttons: on, off, plus, minus, and enter.

Model this problem using

a. A class model

b. An abstract object model.

# 9.3 Behavior modeling

Goal: describe dynamic system behavior

- How the system reacts to a sequence of external events
- How independent system components coordinate their work

Means:

- Finite state machines (FSMs) – not discussed here
- Statecharts / State machines
  - Easier to use than FSMs (although theoretically equivalent)
  - State machines are the UML variant of statecharts
- Sequence diagrams (primarily for behavioral scenarios)
- Petri nets – not discussed here

# Statecharts

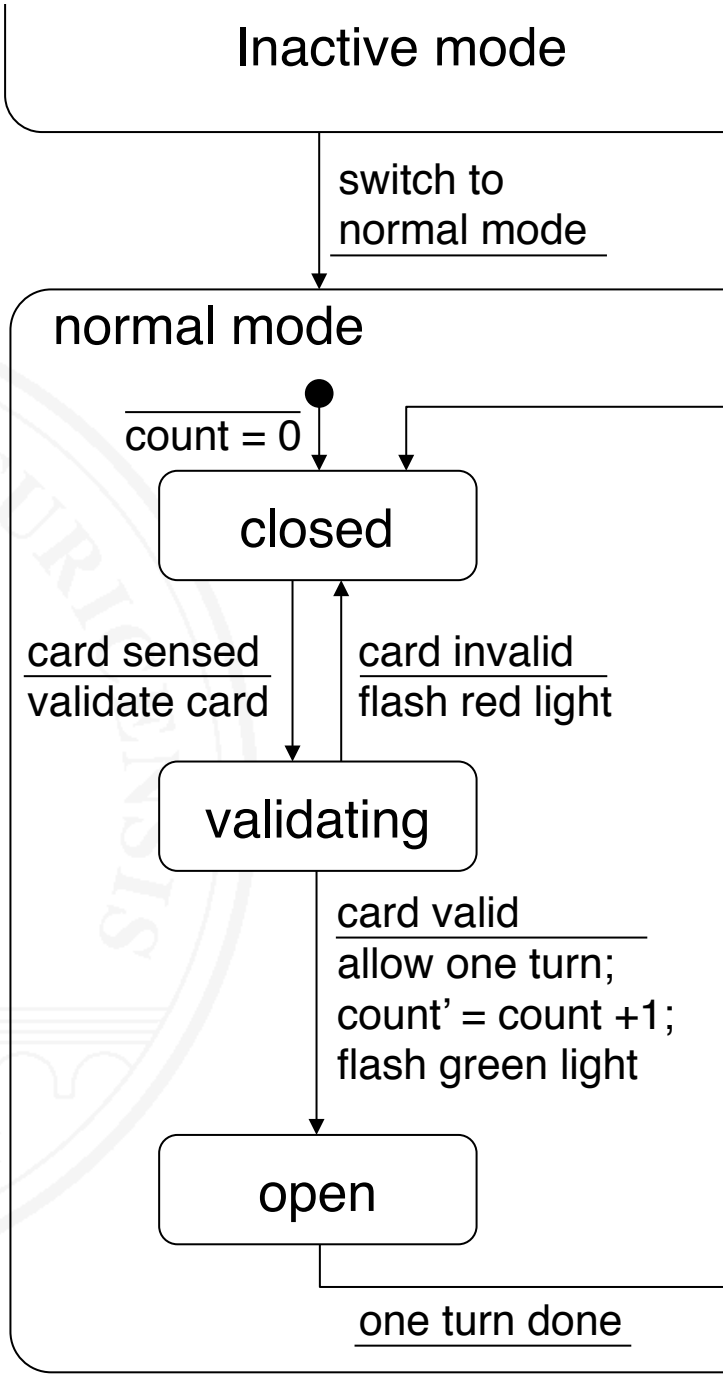
[Harel 1988]

- Models the *dynamic* behavior:
  - How the system reacts to external events in a given state
  - Reaction depends on actual state
  - States may be hierarchically nested and/or orthogonal (parallel)
- In UML: state machine diagrams

\+ Global view of system behavior

\+ Precise, but still readable

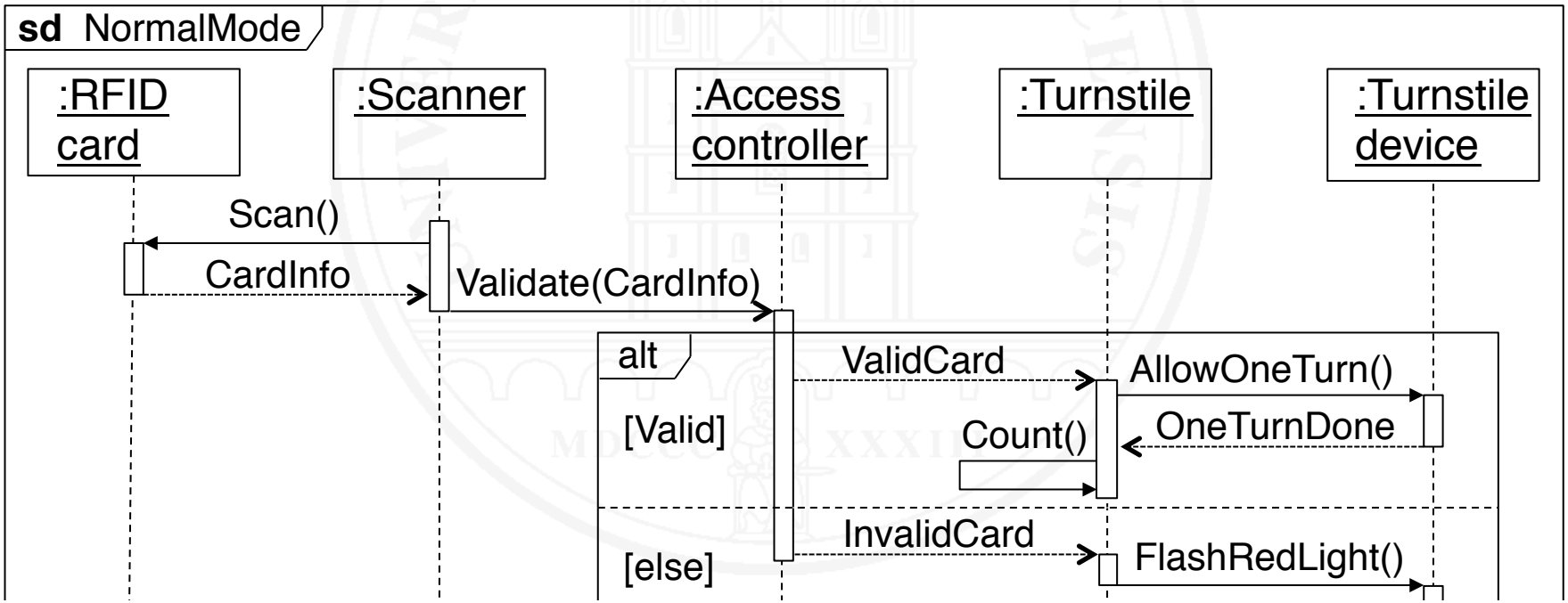
– Weak for modeling functionality and data



# Sequence diagrams / MSCs

Object Management Group (2011b)

- Models ...
  - ... lifelines of system components or objects
  - ... messages that the components exchange



- Notation/terminology:
  - UML: Sequence diagram
  - Otherwise: Message sequence chart (MSC)

\+ Visualizes component collaboration on a timeline

– In practice confined to the description of required scenarios

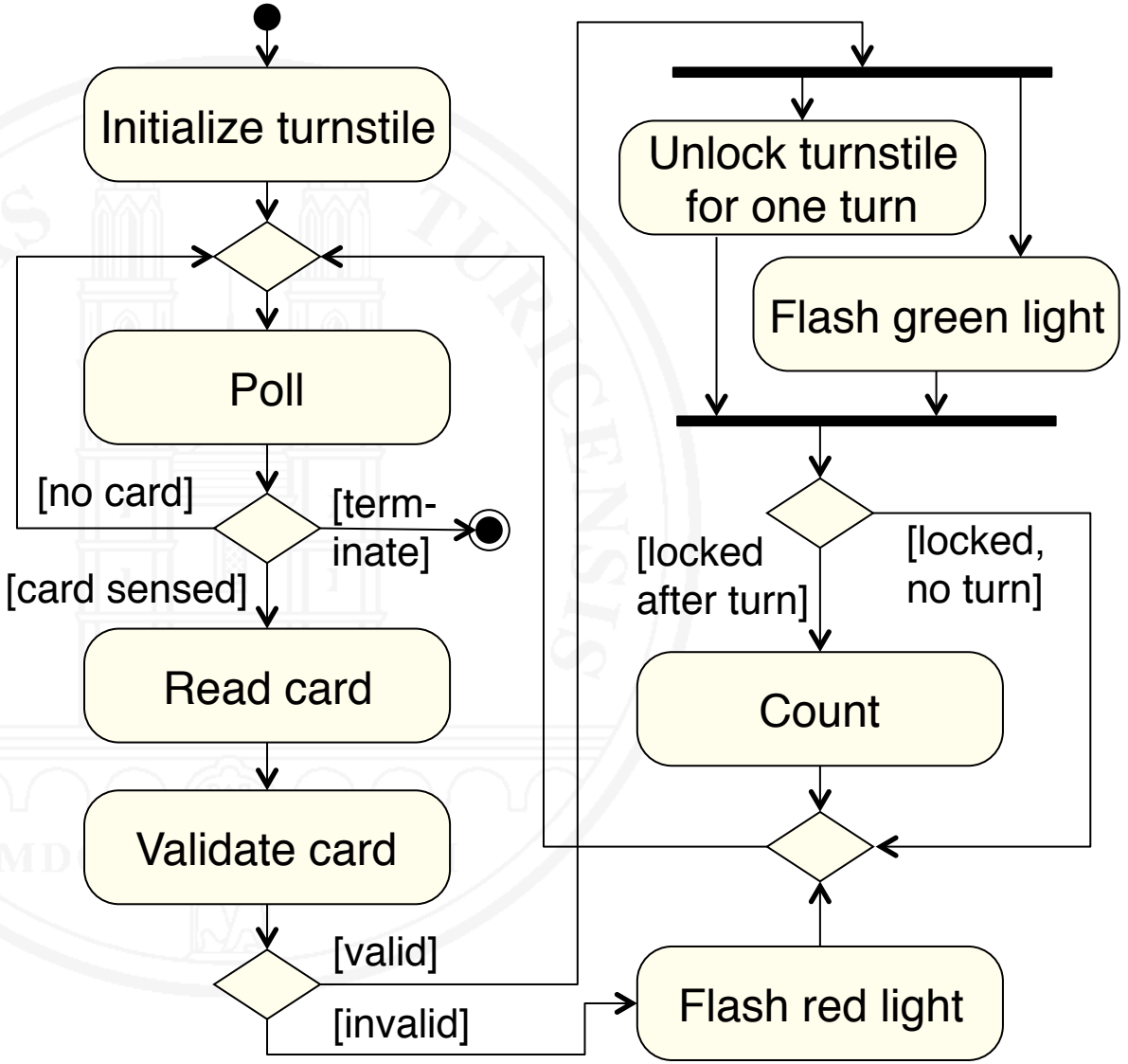
– Design-oriented, can detract from modeling requirements

# 9.4 Function and flow modeling

- Activity models
- Data flow / information flow models
- Process and work flow models

# Activity modeling: UML activity diagram

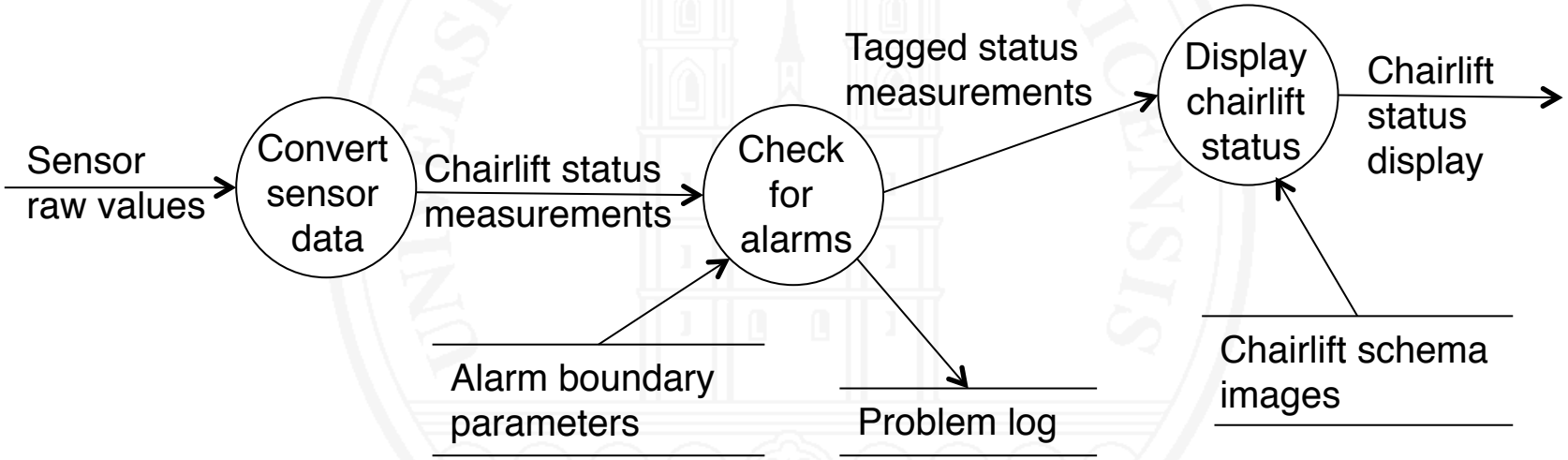
- Models process activities and control flow
- Can model data flow
- Model can be underpinned with execution semantics



# Data and information flow

[DeMarco 1978]

- Models system functionality with data flow diagrams
- Once a dominating approach; rarely used today



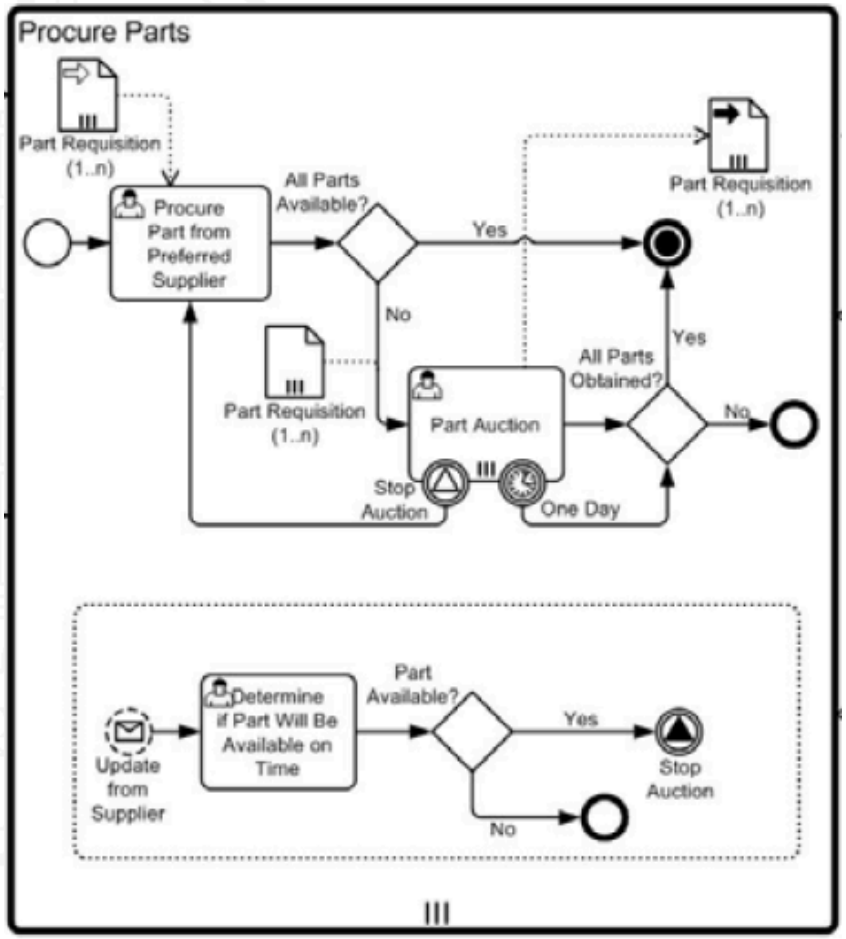

+ Easy to understand
+ Supports system decomposition
– Treatment of data outdated: no types, no encapsulation

# Process and workflow modeling

- Elements
  - Process steps / work steps
  - Events influencing the flow
  - Control flow
  - Maybe data / information access and responsibilities
- Typical languages
  - UML activity diagrams
  - BPMN
  - Event-driven process chains

# Process modeling: BPMN

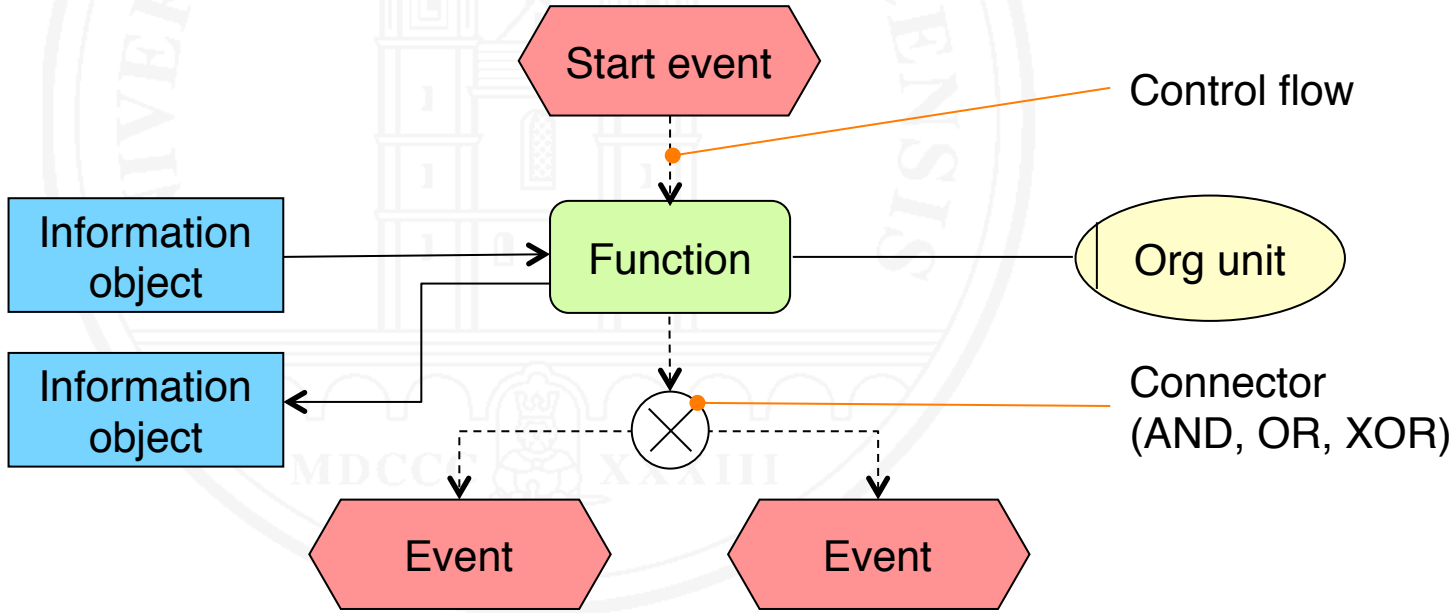
- BPMN (Business Process Model and Notation)
- Rich language for describing business processes




[Object Management Group 2011]

# Process modeling: EPC

- Event-driven process chains (In German: ereignisgesteuerte Prozessketten, EPK)
- Adopted by SAP for modeling processes supported by SAP's ERP software



# 9.5 User-system interaction modeling

Describing the functionality of a system from a user's perspective: How can a user interact with the system?

Two key terms:

- Use case
- Scenario

[Carroll 1995,
Glinz 1995,
Glinz 2000a,
Jacobson et al. 1992,
Sutcliffe 1998,
Weidenhaupt et al. 1998]

# Use case

DEFINITION. Use case – A description of the interactions possible between actors and a system that, when executed, provide added value.

Use cases specify a system from a user's (or other external actor's) perspective: every use case describes some functionality that the system must provide for the actors involved in the use case.

- Use case diagrams provide an overview
- Use case descriptions provide the details

[Jacobson et al. 1992
Glinz 2013]

# Scenario

DEFINITION. Scenario – 1. A description of a potential sequence of events that lead to a desired (or unwanted) result. 2. An ordered sequence of interactions between partners, in particular between a system and external actors. May be a concrete sequence (instance scenario) or a set of potential sequences (type scenario, use case). 3. In UML: An execution trace of a use case.

[Carroll 1995
Sutcliffe 1998
Glinz 1995]

# Use case / scenario descriptions

Various representation options

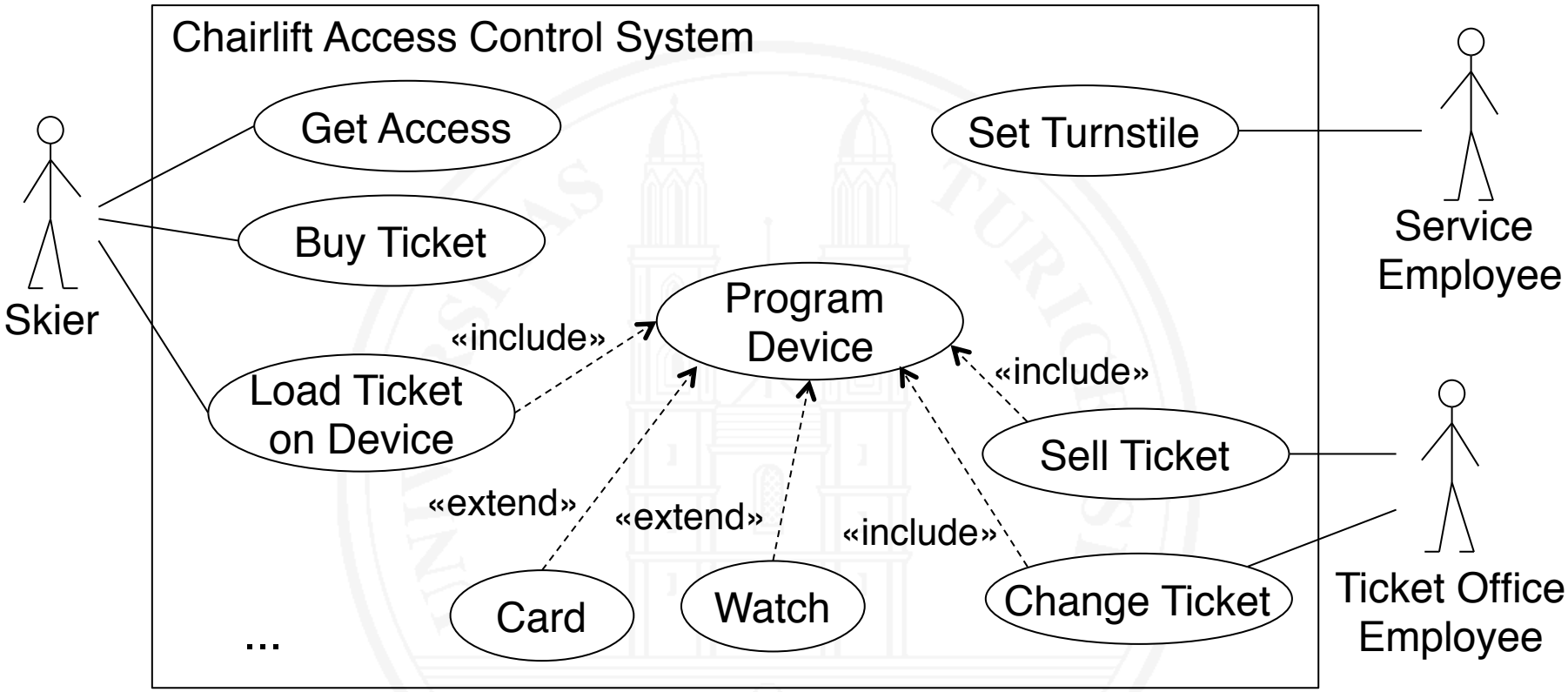
- Free text in natural language
- Structured text in natural language
- Statecharts / UML state machines
- UML activity diagrams
- Sequence diagrams / MSCs

Structured text is most frequently used in practice

# A use case description with structured text

USE CASE SetTurnstiles
Actor: Service Employee
Precondition: none
Normal flow:
1 Service Employee chooses turnstile setup.
System displays controllable turnstiles: locked in red, normal in green, open in yellow.
2 Service Employee selects turnstiles s/he wants to modify.
System highlights selected turnstiles.
3 Service Employee selects Locked, Normal, or Open.
System changes the mode of the selected turnstiles to the selected one, displays all turnstiles in the color of the current mode.
...
Alternative flows:
3a Mode change fails: System flashes the failed turnstile in the color of its current mode.
...

# UML Use case diagram

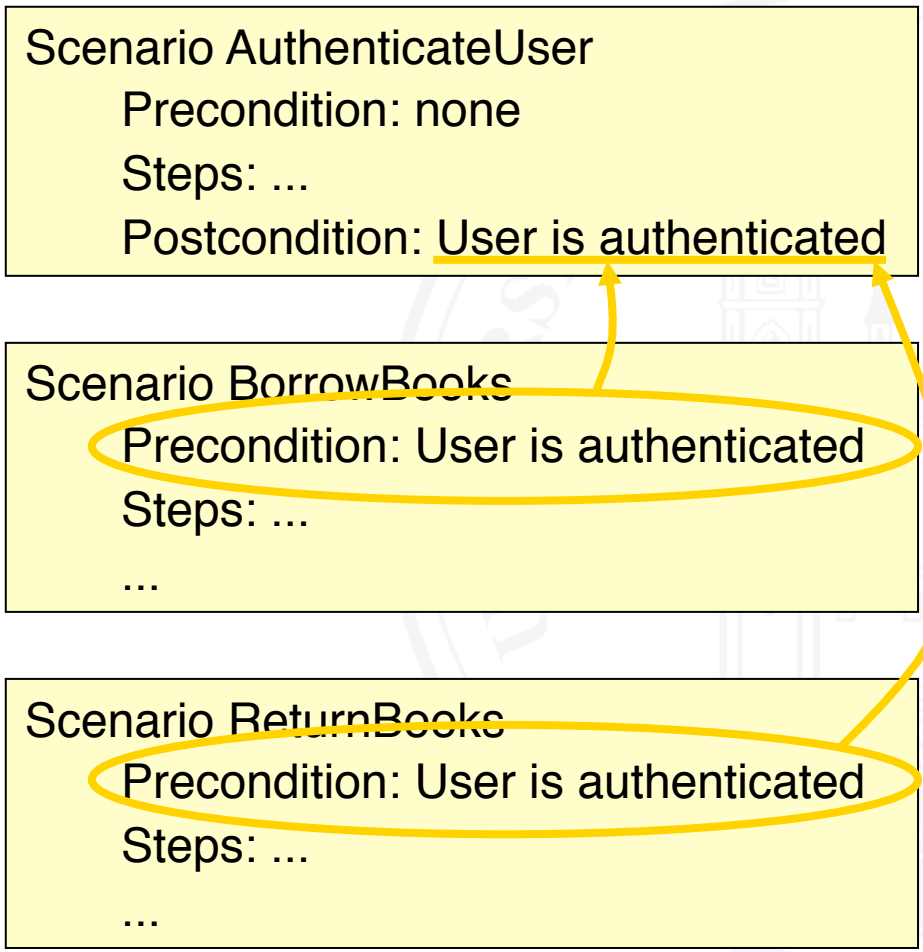



+ Provides abstract overview from actors' perspectives
– Ignores functions and data required to provide interaction
– Can't properly model hierarchies and dependencies

# Dependencies between scenarios / use cases

- UML can only model inclusion, extension and generalization
- However, we need to model
  - Control flow dependencies (sequence, alternative, iteration)
  - Hierarchical decomposition
- Largely ignored in UML (Glinz 2000b)
- Options
  - Pre- and postconditions
  - Statecharts
  - Extended Jackson diagrams (in ADORA, Glinz et al. 2002)
  - Specific dependency charts (Ryser and Glinz 2001)

# Dependencies with pre- and postconditions

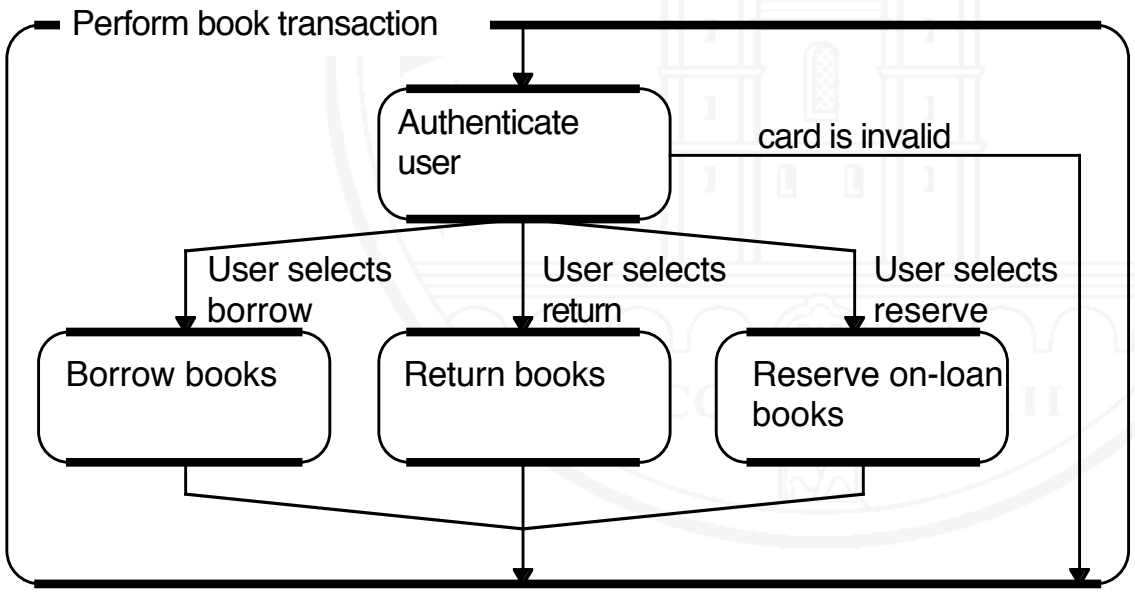



- Simple dependencies of kind «B follows A» can be modeled
- Relationships buried in use case descriptions, no overview
- No hierarchical decomposition
- Modeling of complex relationships very complicated

# Dependencies with Statecharts

[Glinz 2000a]

- Model scenarios as states*
- Classic dependencies (sequence, alternative, iteration, parallelism) can be modeled easily
- Hierarchical decomposition is easy



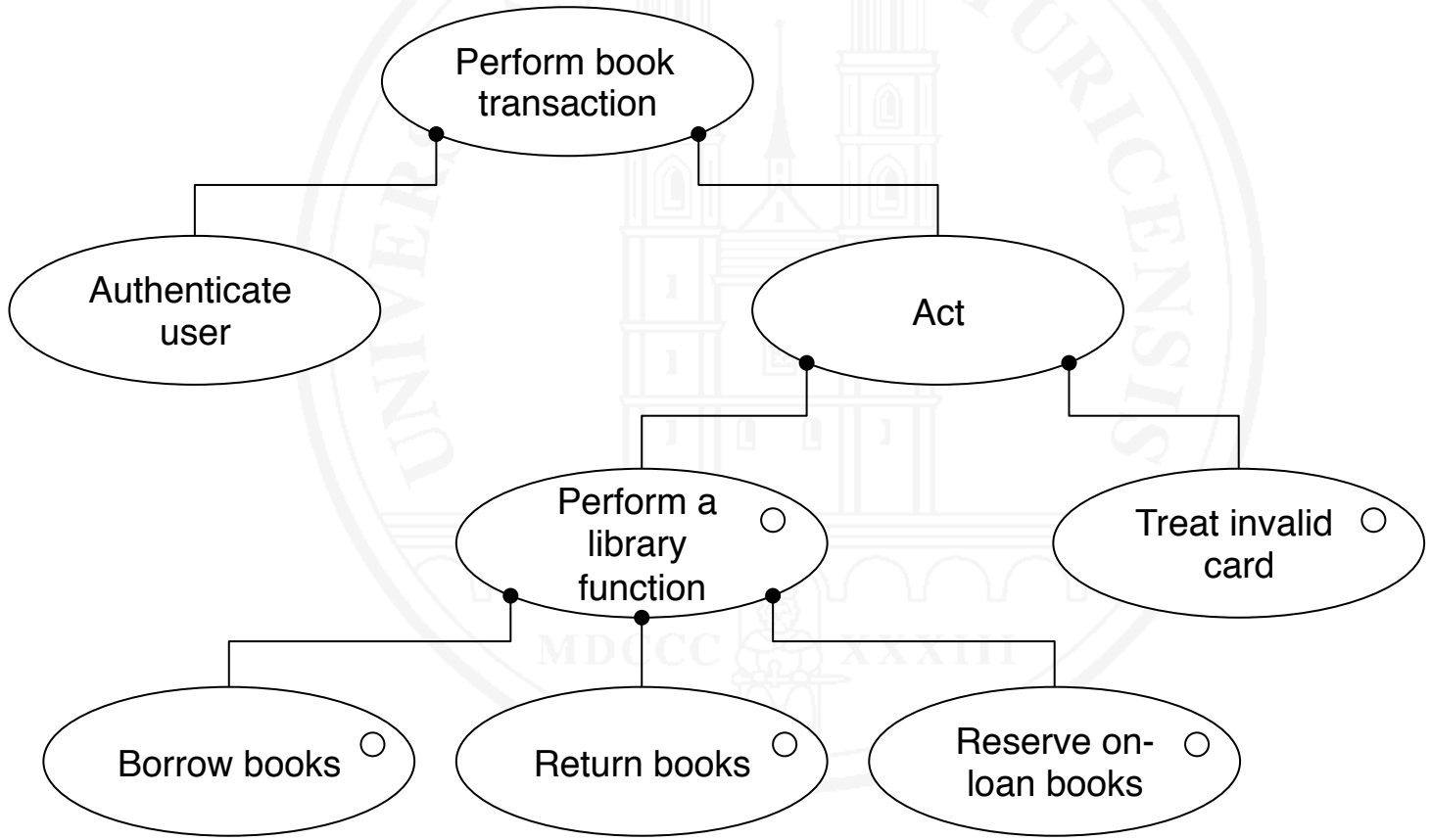

Research result, not used in today's practice

* With one main entry and exit point each; symbolized by top and bottom bars in the diagram

# Dependencies with extended Jackson-diagrams

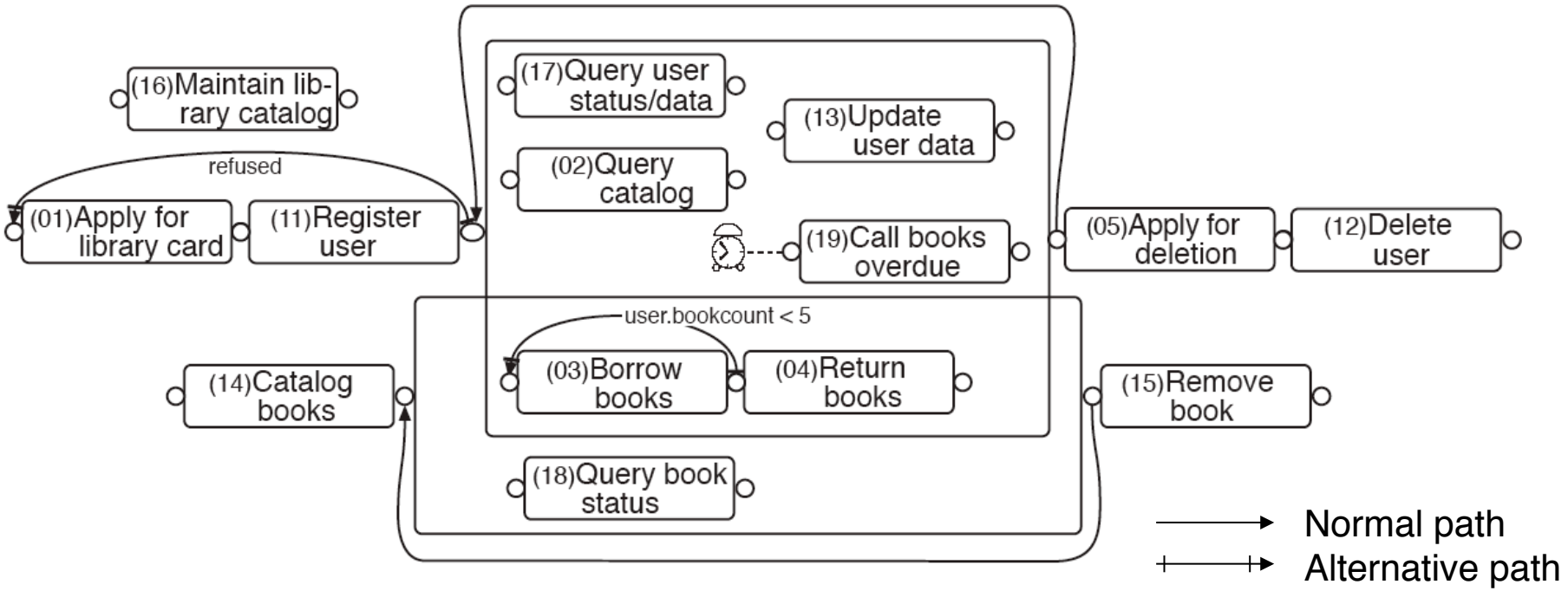
[Glinz et al. 2002]

- Used in ADORA for modeling scenario dependencies



# Dependency charts

- Specific notation for modeling of scenario dependencies (Ryser und Glinz 2001)
- Research result; not used in today's practice



# Mini-Exercise: Writing a use case

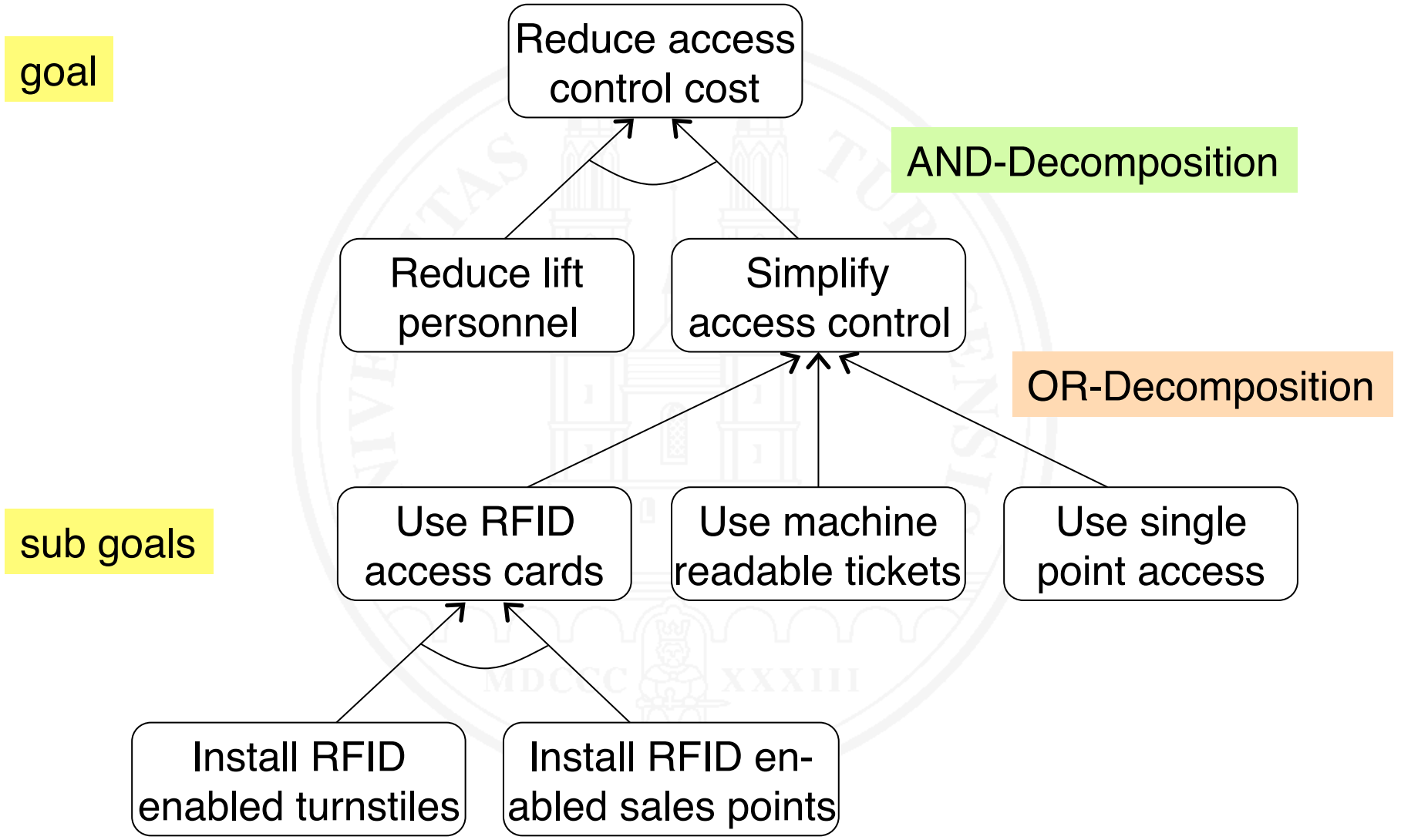
For the Chairlift access control system, write the use case “Get Access”, describing how a skier gets access to a chairlift using his or her RFID ticket.

# 9.6 Modeling goals

- Knowing the goals of an organization (or for a product) is essential when specifying a system to be used in that organization (or product)
- Goals can be decomposed into sub goals
- Goal decomposition can be modeled with AND/OR trees
- Considering multiple goals results in a directed goal graph

[van Lamsweerde 2001, 2004
Mylopoulos 2006
Yu 1997]

# AND/OR trees for goal modeling

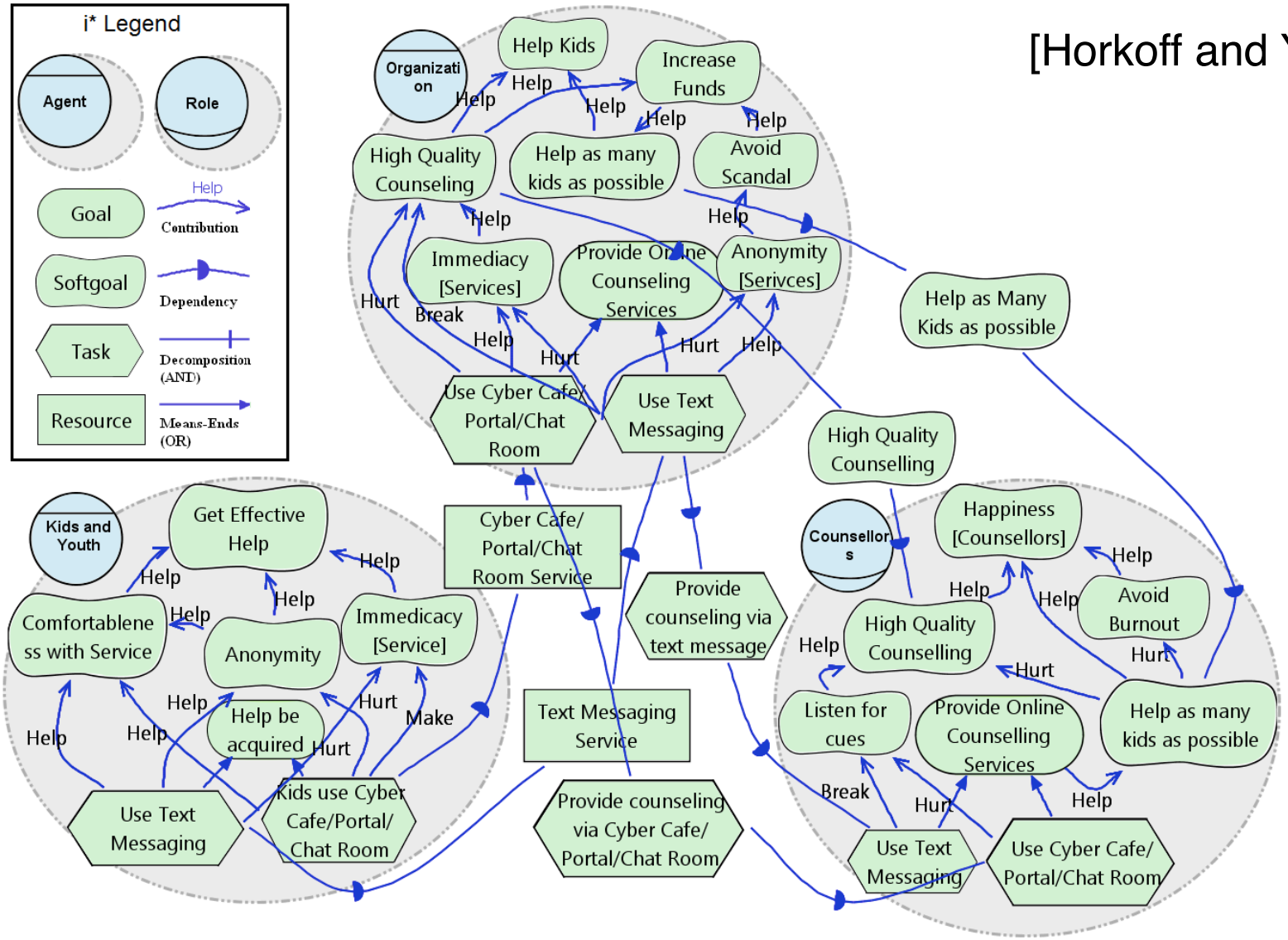


# Goal-agent networks

- Explicitly models agents (stakeholders), their goals, tasks that achieve goals, resources, and dependencies between these items
- Many approaches in the RE literature
- i* is the most popular approach
- Rather infrequently used in practice

# A real world i* example: Youth counseling

[Horkoff and Yu 2010]



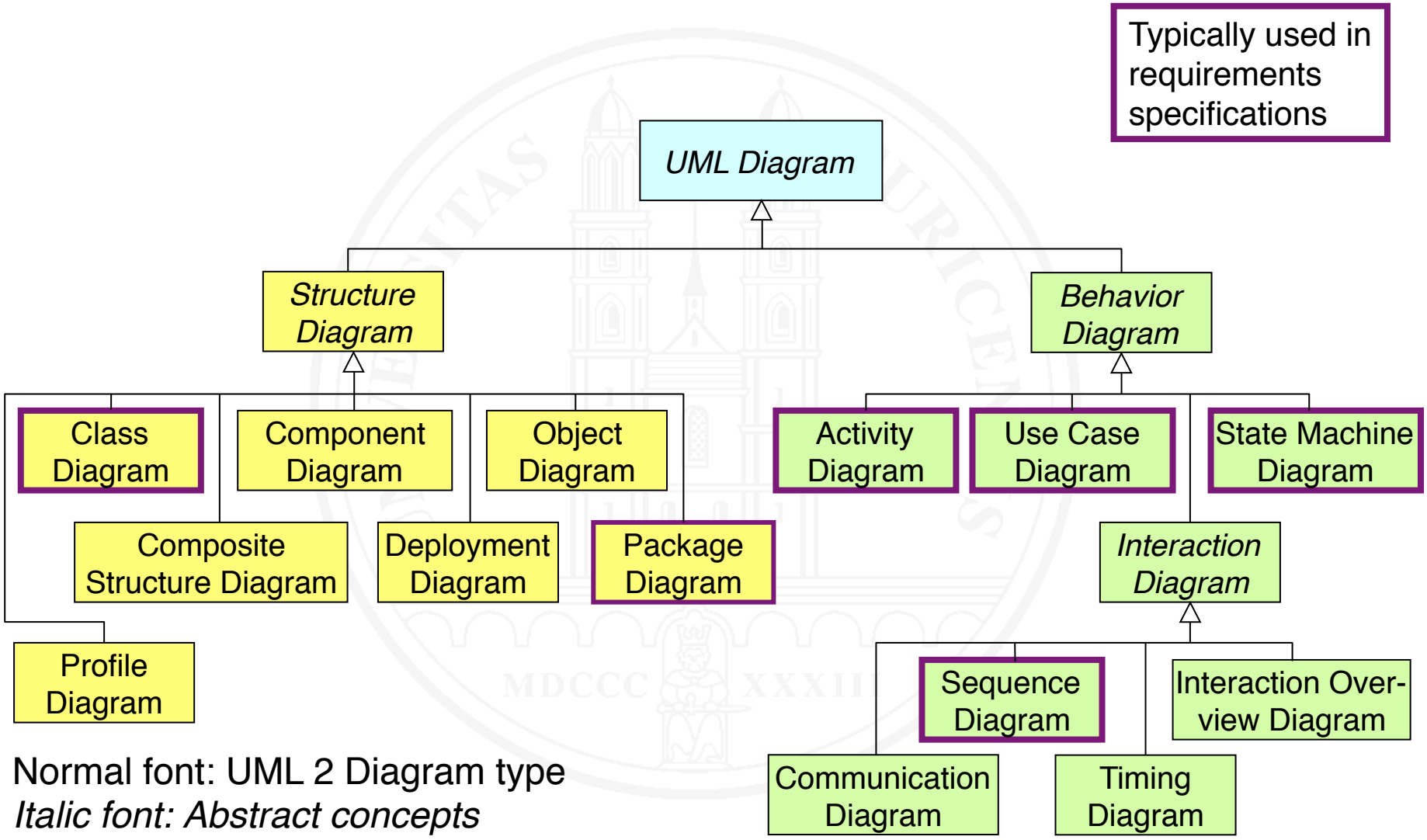
# 9.7 UML (Unified Modeling Language)

[Object Management Group 2015]

- UML is a collection of primarily graphic languages for expressing requirements models, design models, and deployment models from various perspectives
- A UML specification typically consists of a collection of loosely connected diagrams of various types
- Additional restrictions can be specified with the formal textual language OCL (Object Constraint Language)

[Object Management Group 2012]

# UML – Overview of diagram types




Normal font: UML 2 Diagram type
*Italic font: Abstract concepts*

# 10 Formal specification languages

Requirements models with formal syntax and semantics

The vision

- Analyze the problem
- Specify requirements formally
- Implement by correctness-preserving transformations
- Maintain the specification, no longer the code

Typical languages

- “Pure” Automata / Petri nets
- Algebraic specification
- Temporal logic: LTL, CTL
- Set&predicate-based models: Z, OCL, B

# What does “formal” mean?

- Formal calculus, i.e., a specification language with
  - formally defined syntax

    and
  - formally defined semantics
- Primarily for specifying functional requirements

Potential forms

- Purely descriptive, e.g., algebraic specification
- Purely constructive, e.g., Petri nets
- Model-based hybrid forms, e.g. Alloy, B, OCL, VDM, Z

# 10.1 Algebraic specification

- Originally developed for specifying complex data from 1977
- Signatures of operations define the syntax
- Axioms (expressions being always true) define semantics
- Axioms primarily describe properties that are invariant under execution of operations

\+ Purely descriptive and mathematically elegant

– Hard to read

– Over- and underspecification difficult to spot

– Has never made it from research into industrial practice

# Algebraic specification: a simple example

Specifying a stack (last-in-first-out) data structure

Let bool be a data type with a range of {false, true} and boolean algebra as operations. Further, let elem be the data type of the elements to be stored.

```
TYPE Stack
FUNCTIONS
new:    ()              →  Stack;   -- Create new (empty) stack
push:   (Stack, elem)   →  Stack;   -- add an element
pop:    Stack           →  Stack;   -- remove most recent element from stack
top:    Stack           →  elem;    -- returns most recent element
empty:  Stack           →  bool;    -- true if stack is empty
full:   Stack           →  bool;    -- true if stack is full
```

# Algebraic specification: a simple example – 2

AXIOMS

$\forall s \in Stack, e \in elem$

| | | |
|---|---|---|
| (1) | $\neg$ full(s) $\rightarrow$ pop(push(s,e)) = s | -- *pop* reverses the effect of push |
| (2) | $\neg$ full(s) $\rightarrow$ top(push(s,e)) = e | -- *top* retrieves the most recently stored element |
| (3) | empty(new) = true | -- a *new* stack is always empty |
| (4) | $\neg$ full(s) $\rightarrow$ empty(push(s,e)) = false | -- after *push*, a stack is not empty |
| (5) | full(new) = false | -- a *new* stack is not full |
| (6) | $\neg$ emtpy(s) $\rightarrow$ full(pop(s)) = false | -- after *pop*, a stack is not full |

# 10.2 Model-based formal specification

- Mathematical model of system state and state change
- Based on sets, relations and logic expressions
- Typical language elements
  - Base sets
  - Relationships (relations, functions)
  - Invariants (predicates)
  - State changes (by relations or functions)
  - Assertions for states

# The formal specification language landscape

- VDM – Vienna Development Method (Björner and Jones 1978)
- Z (Spivey 1992)
- OCL (from 1997; OMG 2012)
- Alloy (Jackson 2002)
- B (Abrial 2009)

# 10.3 An overview of Z

- A typical model-based formal language
- Only basic concepts covered here
- More detail in the literature, e.g., Jacky (1997)

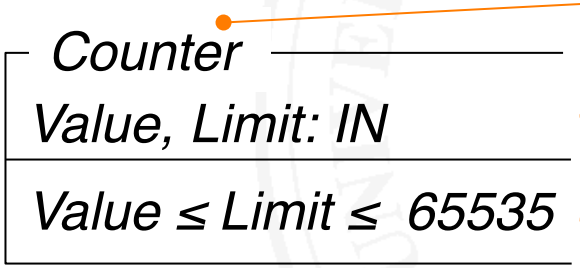
# The basic elements of Z

- Z is set-based
- Specification consists of sets, types, axioms and schemata
- Types are elementary sets: *[Name]* *[Date]* *IN*
- Sets have a type: *Person: ℙ Name* *Counter: IN*
- Axioms define global variables and their (invariant) properties


| | |
|---|---|
| *IN* | Set of natural numbers |
| *ℙ M* | Power set (set of all subsets) of *M* |
| ***seq*** | Sequence of elements |
| *#M* | Number of elements of set *M* |

# The basic elements of Z – 2

- Schemata
  - organize a Z-specification
  - constitute a name space




Name

Declaration part:
Declaration of state variables

Predicate part:
- Restrictions
- Invariants
- Relationships
- State change

# Relations, functions und operations

- Relations and functions are ordered set of tuples:

  *Order: ℙ (Part x Supplier x Date)*

  *Birthday: Person → Date*

A subset of all ordered triples (p, s, d) with p ∈ *Part*, s ∈ *supplier,* and d ∈ *Date*

A function assigning a date to a person, representing the person's birthday

State change through operations:

*Increment counter*

*Δ Counter*

---

*Value < Limit*
*Value' = Value + 1*
*Limit' = Limit*

| | |
|---|---|
| *Δ S* | The sets defined in schema S will be changed |
| *M'* | State of set M after executing the operation |

Mathematical equality, no assignment!

# Example: specification of a library system

The library has a stock of books and a set of persons who are library users.

Books in stock may be borrowed.

*Library*

$$Stock: \mathbb{P}\ Book$$
$$User: \mathbb{P}\ Person$$
$$lent: Book \pfun Person$$

---

$$\mathbf{dom}\ lent \subseteq Stock$$
$$\mathbf{ran}\ lent \subseteq User$$

| | |
|---|---|
| $\pfun$ | Partial function |
| ***dom*** | Domain ... |
| ***ran*** | Range... |
| | ...of a relation |

# Example: specification of a library system – 2

Books in stock which currently are not lent to somebody may be borrowed

*Borrow*

$\Delta$ *Library*
*BookToBeBorrowed?: Book*
*Borrower?: Person*

---

*BookToBeBorrowed?* $\in$ *Stock* \ ***dom*** *lent*
*Borrower?* $\in$ *User*
*lent'* = *lent* $\cup$ {(*BookToBeBorrowed?*, *Borrower?*)}
*Stock'* = *Stock*
*User'* = *User*

| | |
|---|---|
| *x?* | *x* is an input variable |
| $a \in X$ | *a* is an element of set *X* |
| \ | Set difference operator |
| $\cup$ | Set union operator |

# Example: specification of a library system – 3

It shall be possible to inquire whether a given book is available

*InquireAvailability*

$\Xi$ *Library*
*InquiredBook?: Book*
*isAvailable!: {yes, no}*

---

*InquiredBook?* $\in$ *Stock*
*isAvailable!* = ***if*** *InquiredBook?* $\notin$ ***dom*** *lent*
***then*** *yes* ***else*** *no*

| | |
|---|---|
| $\Xi S$ | The sets defined in schema S can be referenced, but not changed |
| *x!* | x is an output variable |

# Mini-Exercise: Specifying in Z

Specify a system for granting and managing authorizations for a set of individual documents.

The following sets are given:

*Authorization*

$Stock : \mathbb{P}\ Document$
$Employee : \mathbb{P}\ Person$
$authorized : \mathbb{P}\ (Document \times Person)$
$prohibited : \mathbb{P}\ (Document \times Date)$

Specify an operation for granting an employee access to a document as long as access to this document is not prohibited. Use a Z-schema.

.

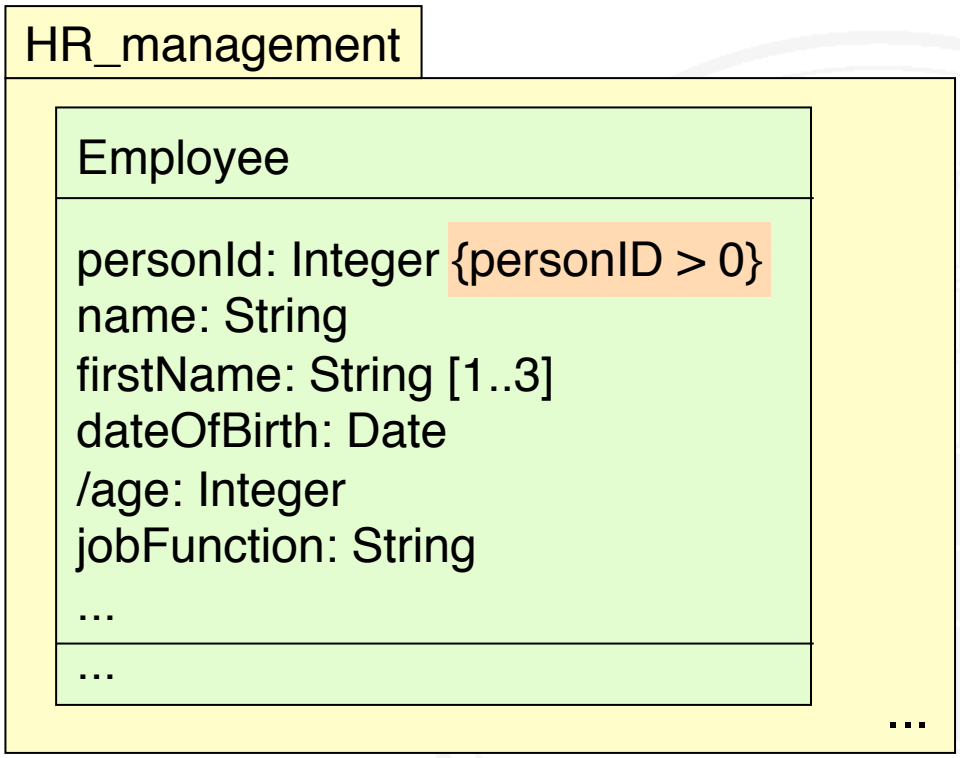
# 10.4 OCL (Object Constraint Language)

- What is OCL?
  - A textual formal language
  - Serves for making UML models more precise
  - Every OCL expression is attached to an UML model element, giving the context for that expression
  - Originally developed by IBM as a formal language for expressing integrity constraints (called ICL)
  - In 1997 integrated into UML 1.1
  - Current standardized version is Version 2.3.1
  - Also published as an ISO standard: ISO/IEC 19507:2012

# Why OCL?

- Making UML models more precise
  - Specification of Invariants (i.e., additional restrictions) on UML models
  - Specification of the semantics of operations in UML models
- Also usable as a language to query UML models

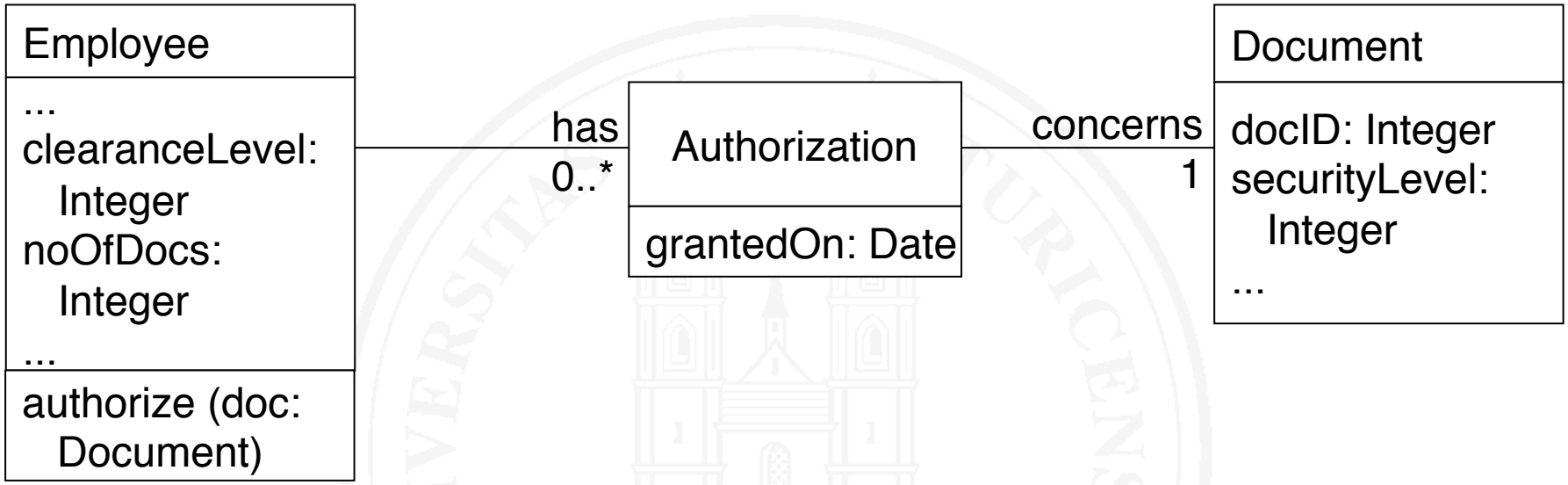
# OCL expressions: invariants




**context** HR_manangement::Employee **inv**:
self.jobFunction = “driver” **implies** self.age ≥ 18

- OCL expression may be part of a UML model element
- Context for OCL expression is given implicitly
- OCL expression may be written separately
- Context must be specified explicitly

# OCL expressions: Semantics of operations




**context** Employee::authorize (doc: Document)
**pre**: self.clearanceLevel ≥ doc.securityLevel
**post**: noOfDocs = noOfDocs@pre + 1
**and**
self.has->**exists** (a: Authorization | a.concerns = doc)

# Navigation, statements about sets in OCL

- Persons having Clearance level 0 can't be authorized for any document:

  **context** Employee **inv**: self.clearanceLevel = 0 **implies** self.has->isEmpty()

Navigation from current object to a set of associated objects

Application of a function to a set of objects

# Navigation, statements about sets in OCL – 2

More examples:

- The number of documents listed for an employee must be equal to the number of associated authorizations:

  **context** Employee **inv**: self.has->size() = self.noOfDocs

- The documents authorized for an employee are different from each other

  **context** Employee **inv**: self.has->**forAll** (a1, a2: Authorization |
  a1 <> a2 **implies** a1.concerns.docID <> a2.concerns.docID)

- There are no more than 1000 documents:

  **context** Document **inv**: Document.allInstances()->size() ≤ 1000

# Summary of important OCL constructs

- Kind and context: **context**, **inv**, **pre**, **post**
- Boolean logic expressions: **and**, **or**, **not**, **implies**
- Predicates: **exists**, **forAll**
- Alternative: **if then else**
- Set operations: size(), isEmpty(), notEmpty(), sum(), ...
- Model reflection, e.g., *self.oclIsTypeOf (Employee)* is true in the context of Employee
- Statements about all instances of a class: allInstances()
- Navigation: dot notation — self.has.date = ...
- Operations on sets: arrow notation — self.has->size()
- State change: @pre notation — noOfDocs = noOfDocs@pre + 1

# 10.5 Proving properties

With formal specifications, we can prove if a model has some required properties (e.g., safety-critical invariants)

- Classic proofs (usually supported by theorem proving software) establish that a property can be inferred from a set of given logical statements
- Model checking explores the full state space of a model, demonstrating that a property holds in every possible state

– Classic proofs are still hard and labor-intensive

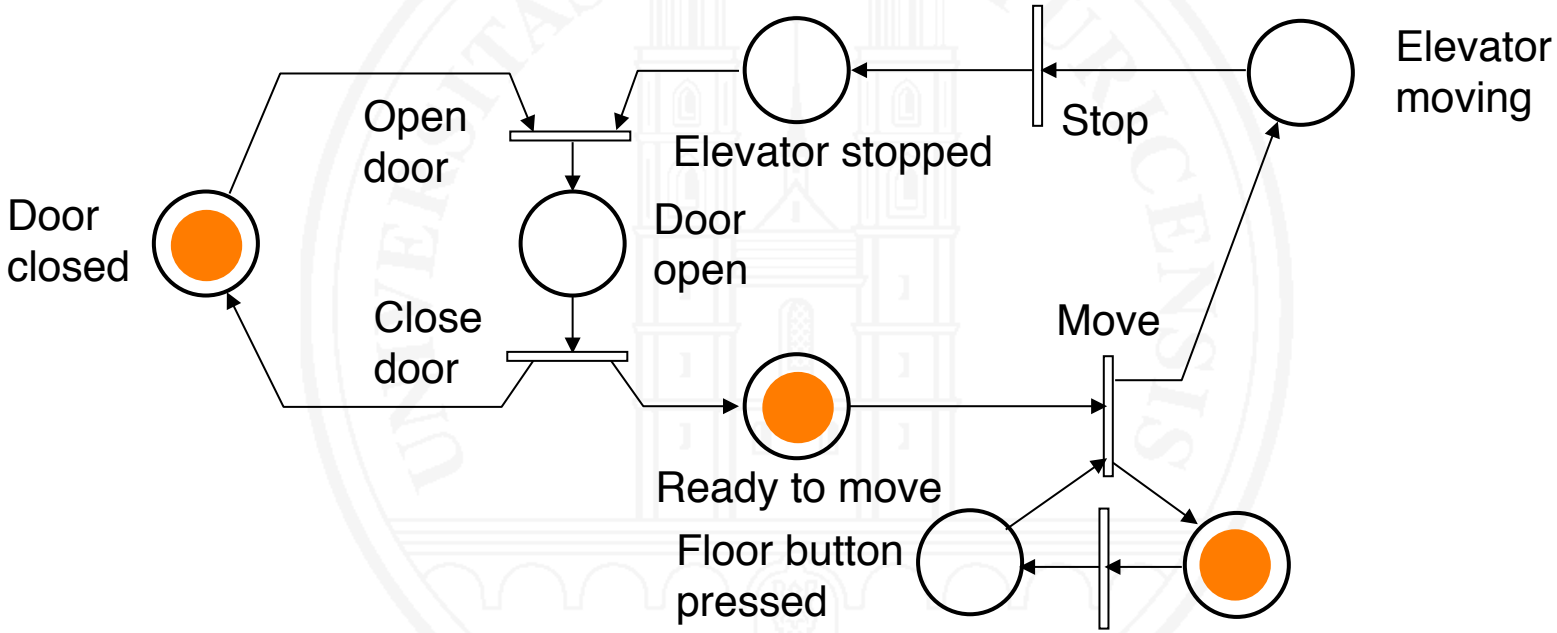
\+ Model checking is fully automatic and produces counter-examples in case of failure

– Exploring the full state state space is frequently infeasible

\+ Exploring feasible subsets is a systematic, automated test

# Example: Proving a safety property

A (strongly simplified) elevator control system has been modeled with a Petri net as follows:




The property that an elevator never moves with doors open shall be proved

# Example: Proving a safety property – 2

The property to be proven can be restated as:

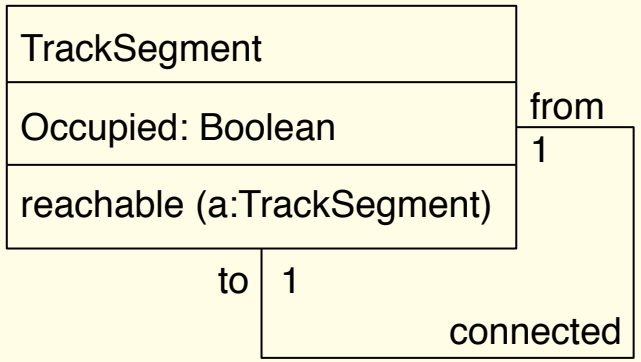
(P) The places *Door open* and *Elevator moving* never hold tokens at the same time

Due to the definition of elementary Petri Nets we have

- The transition *Move* can only fire if *Ready to move* has a token (1)
- There is at most one token in the cycle *Ready to move* – *Elevator moving* – *Elevator stopped* – *Door open* (2)
- (2) ⇒ If *Ready to move* has a token, *Door open* hasn't one (3)
- (2) ⇒ If *Elevator moving* has a token, *Door open* hasn't one (4)
- If *Door open* has no token, *Door closed* must have one (5)
- (1) & (3) & (4) & (5)⇒ (P) □

# Mini-Exercise: A circular metro line

A circular metro line with 10 track segments has been modeled in UML and OCL as follows:




**Context** TrackSegment::
reachable (a: TrackSegment): Boolean
**post**:
result = (self.to = a) **or** (self.to.reachable (a))

**context** TrackSegment **inv**:
TrackSegment.allInstances->size = 10

In a circle, every track segment must be reachable from every other track segment (including itself). So we must have:

**context** TrackSegment **inv** (1)
TrackSegment.allInstances->forAll (x, y | x.reachable (y) )

a) Falsify this invariant by finding a counter-example

# Mini-Exercise: A circular metro line – 2

Only the following trivial invariant can be proved:

**context** TrackSegment **inv**:
TrackSegment.allInstances->forAll (x | x.reachable (x) )

b) Prove this invariant using the definition of *reachable*

Obviously, this model of a circular metro line is wrong. The property of being circular is not mapped correctly to the model.

c) How can you modify the model such that the original invariant (1) holds?

# 10.6 Benefits and limitations, practical use

Benefits

- Unambiguous by definition
- Fully verifiable
- Important properties can be
  - proven
  - or tested automatically (model checking)

Limitations / problems

- Cost vs. value
- Stakeholders can't read the specification: how to validate?
- Primarily for functional requirements

# Role of formal specifications in practice

- Marginally used in practice
  - Despite its advantages
  - Despite intensive research (research on algebraic specifications dates back to 1977)
- Actual situation today
  - Punctual use possible and reasonable
  - In particular for safety-critical components
  - However, broad usage
    - not possible (due to validation problems)
    - not reasonable (cost exceeds benefit)
- Another option: semi-formal models where critical parts are fully formalized