

Slides for
Design Methods for Reactive Systems:
Yourdon, Statemate and the UML

Roel Wieringa
Department of Computer Science
University of Twente,
the Netherlands
roelw@cs.utwente.nl
www.cs.utwente.nl/~roelw

List of Slides

- 3 Chapter 1. Reactive Systems
- 11 Chapter 2. The Environment
- 28 Chapter 3. Stimulus-Response Behavior
- 43 Chapter 4. Software Specifications
- 55 Chapters 5–7. Mission Statement, Function Refinement Tree, Service Description
- 77 Chapter 8. Entity-Relationship Diagrams
- 99 Chapter 9. ERD Modeling Guidelines
- 125 Chapter 10. The Dictionary
- 139 Part IV. Behavior Notations
- 154 Chapter 11. State Transition Lists and Tables
- 166 Chapter 12. State Transition Diagrams

188	Chapter 13. Behavioral Semantics
210	Chapter 14. Behavior Modeling and Design Guidelines
238	Part V. Communication Notations
246	Chapter 15. Data Flow Diagrams
265	Chapter 16. Communication Diagrams
276	Chapter 17. Communication Semantics
287	Chapter 18. Context Modeling Guidelines
300	Chapter 19. Requirements-Level Decomposition Guidelines
323	Chapter 20. Postmodern Structured Analysis (PSA)
332	Chapter 21. Statemate
351	Chapter 22. The Unified Modeling Language (UML)
381	Chapter 23. Not Yet Another Method

Chapter 1. Reactive Systems

- Partitioning of systems into information systems, control systems and telecommunication systems is becoming obsolete.
- More informative partitioning: Transformational versus reactive systems.
- Reactive systems respond to stimuli in order to bring about desirable effects in their environment.
- Reactive systems may do one or or of these things:
 - Manipulate complex data,
 - engage in complex behavior,
 - communicate with many other systems.

Example 1

A training information system supports coordination of monthly introductory training courses for new employees of large company.

- *Nonterminating*: When switched on, it should respond to events such as queries, updates, arrival of downloads.
- *Interactive*: When switched on, it can engage in dialogs with its users and with other software (personnel information system).
- *State-dependent response*: Its response depends upon the data stored in it.
- *Environment-oriented response*: Responses defined in terms of courses, participants, teachers etc.
- *Parallel processing*: May interact with several users and programs simultaneously.

Example 2

An electronic ticket system supports buying and using rail tickets by means of a smart card in combination with a PDA.

- *Nonterminating*: When switched on, it should respond to events such as buy, show, use.
- *Interactive*: When switched on, it can engage in dialogs with its users and with other software.
- *State-dependent response*: Its response depends upon the tickets and the data about railroads stored in it.
- *Environment-oriented response*: Responses defined in terms of railroad routes and segments.
- *Parallel processing*: It may consist of several pieces of software running concurrently on smart card and PDA.

Example 3

A heating controller of heating tank in juice plant.

- *Nonterminating*: When switched on, it should respond to events such as start heating, too hot, too cold.
- *Interactive*: When switched on, it can engage in dialogs with its users and with devices.
- *Interrupt-driven*: It responds to time-outs and signals from operator and devices as and when they occur.
- *State-dependent response*: Its response depends upon the data that it stores, which is about the heating tank and its devices.
- *Environment-oriented response*: Responses are about heating tank and its devices.
- *Parallel processing*: Monitoring temperature, monitoring pressure, listening to commands from operator.
- *Real time*: Responses would be incorrect if too late.

Definition of reactive system

A **reactive system** is a system that, when switched on, is able to create desired effects in its environment by enabling, enforcing or preventing events in the environment.

Has most of the following characteristics:

- nonterminating
- interactive
- interrupt-driven
- state-dependent
- environment-oriented
- parallel
- real-time

Examples of reactive systems

- Information systems
- Workflow systems
- Groupware
- EDI systems
- Web market places
- Production control software
- Embedded software

Transformational systems

Contrast reactive systems with transformational systems, that compute output from an input and then terminate.

- terminating
- sometimes interactive
- not interrupt-driven
- output not state-dependent
- output defined in terms of input
- sequential
- usually not real-time

Compiler, assembler, loader, expert system, optimization algorithm, search algorithm, linear programming algorithm, etc.

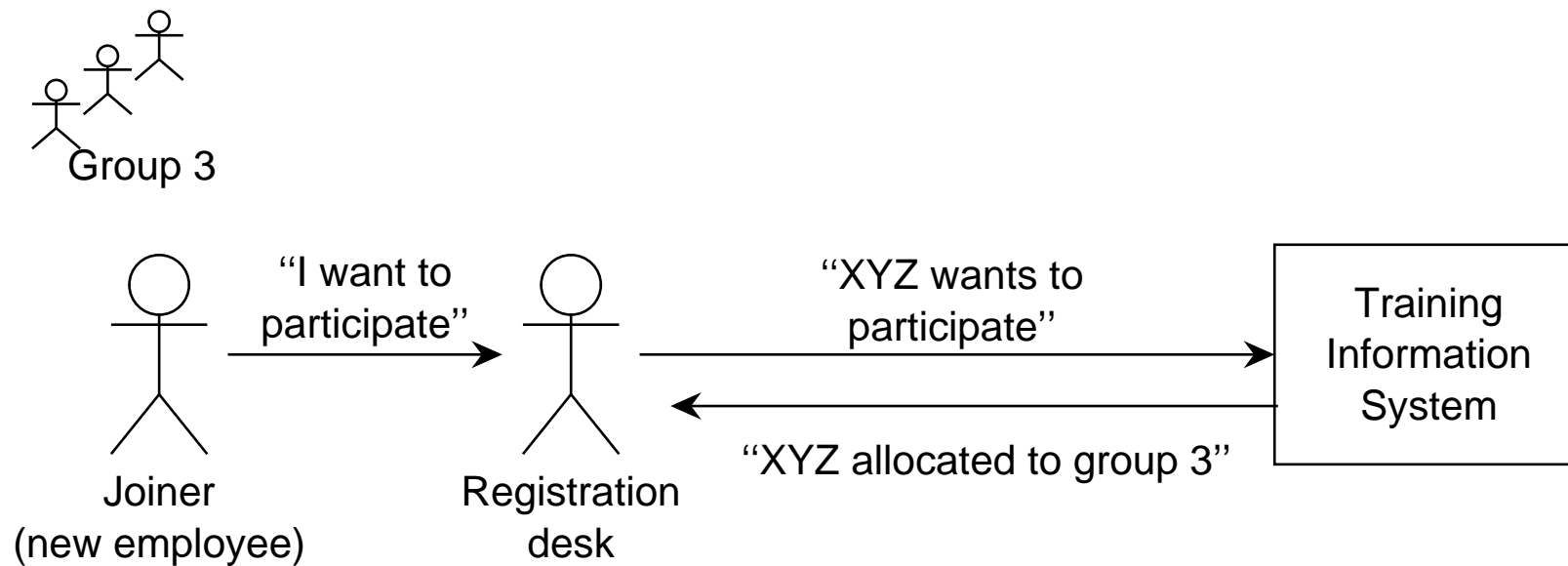
Design approach to reactive systems

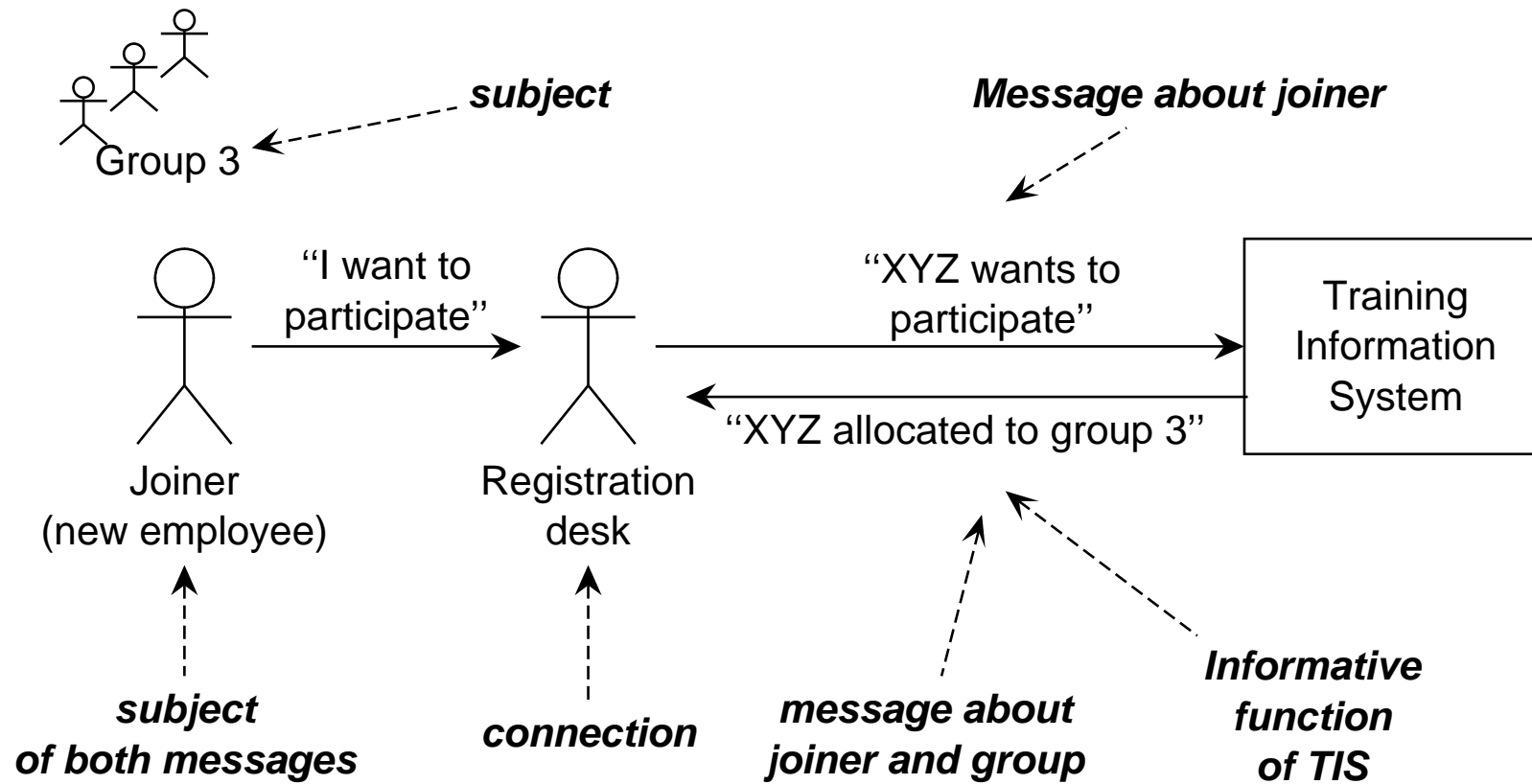
To design reactive systems, environment models are important:

- Possible entities and behavior in environment (chapter 2).
- Communication of system with environment (chapter 3)

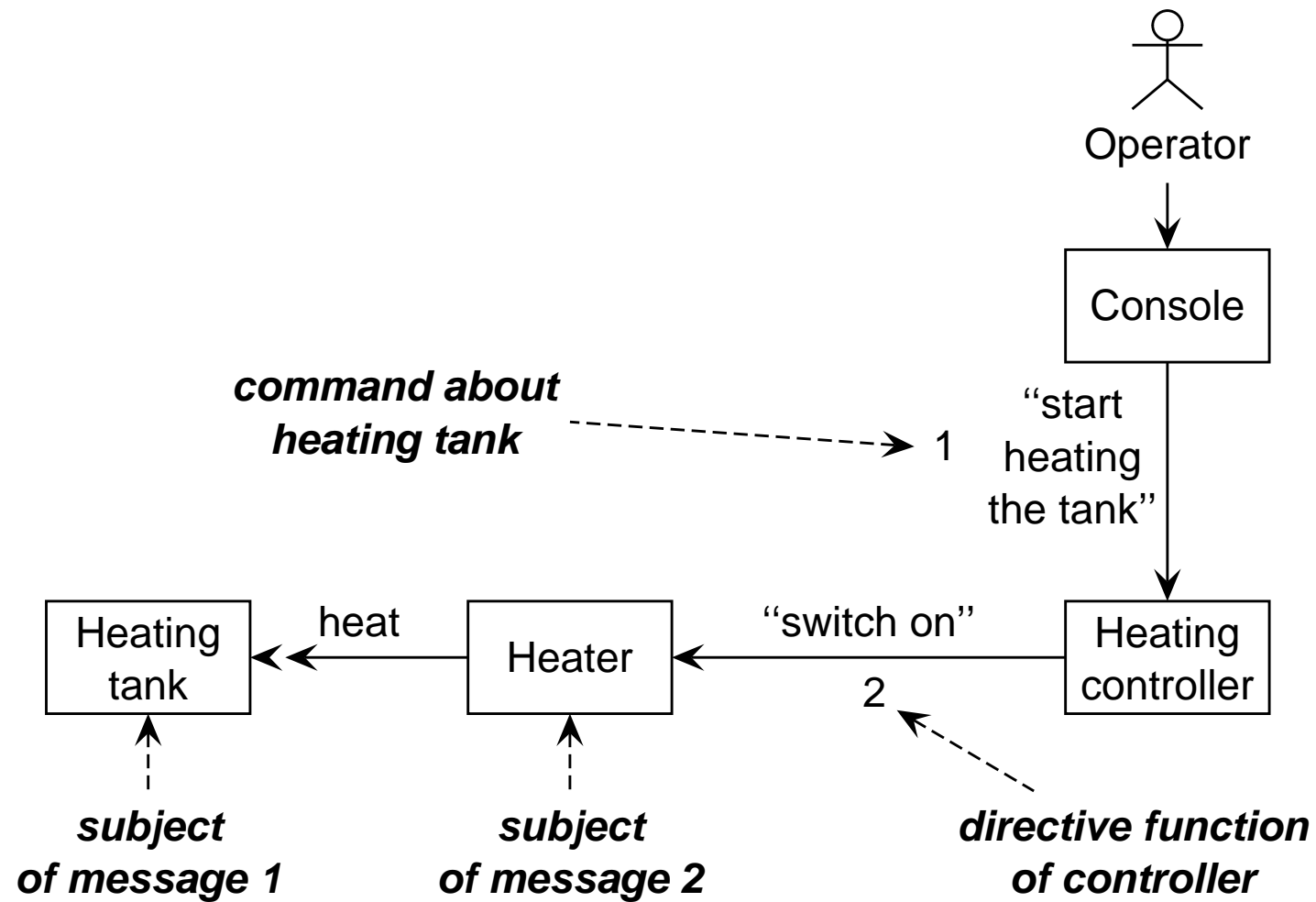
Chapter 2. The Environment

Example 1

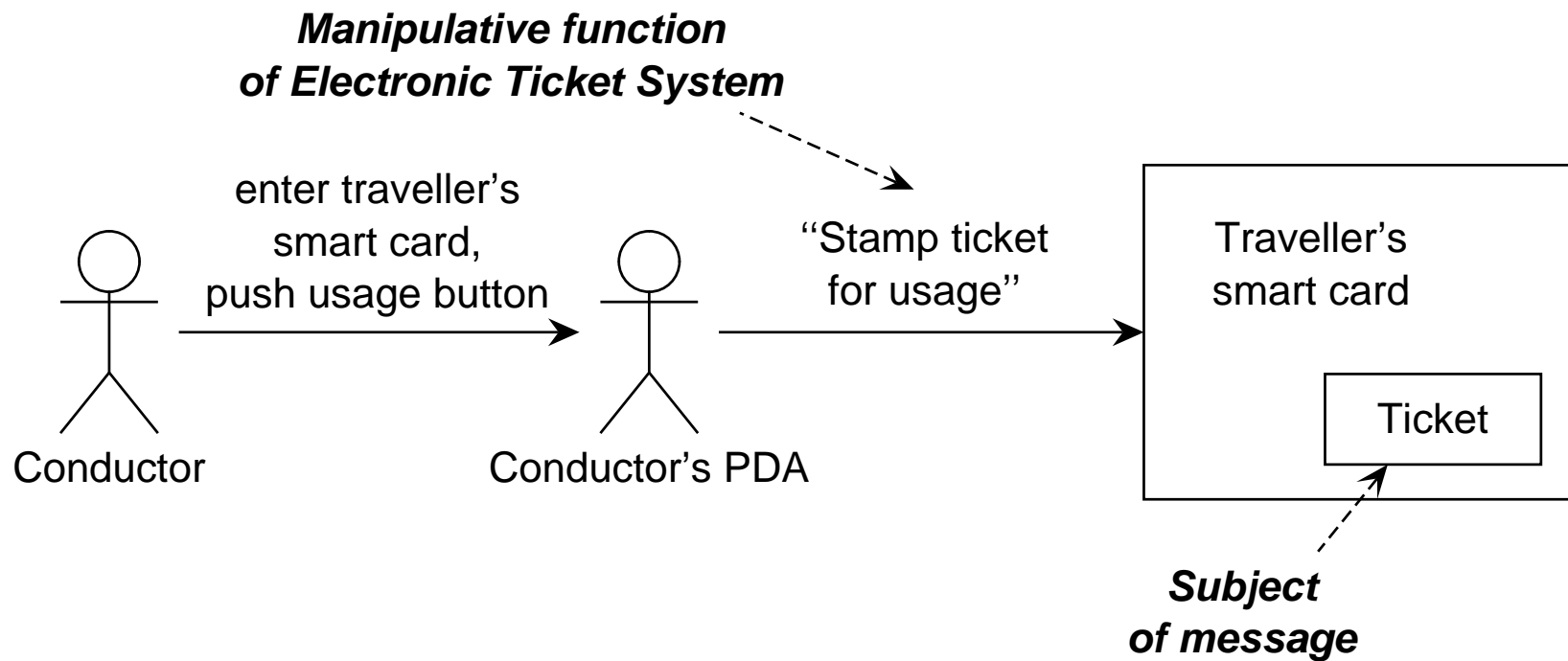




Example 2



Example 3



Symbolic interactions

Reactive software systems communicate with their environment by means of linguistic messages.

- Not in object-oriented sense.
- Post-it notes exchanged with environment.
- Flow of symbols is more important than of flow of energy or matter.
- Physical connection is what makes symbol flow possible.
- Effect not determined by physical causality but by symbol flow.
- We abstract from the physical realization of these messages.

Summary of examples

Messages entering and leaving a reactive software system are characterized by three aspects:

- **A subject**

- What is the message about?

About people, devices, conceptual entities, lexical entities.

- **A function**

- What is the purpose of the message?

To inform or direct the environment, to manipulate lexical items in the system.

- **A connection**

- Through which path does the message travel from sender to receiver?

Messages can get delayed, distorted or lost along the way.

Definitions

- **Subject domain** of a reactive system: Set of all subjects of all its input and output messages.
- **Functions** of a reactive system: All services provided by these message exchanges.
- **Connection domain** of a reactive system: Channels through which messages flow to and from the system.

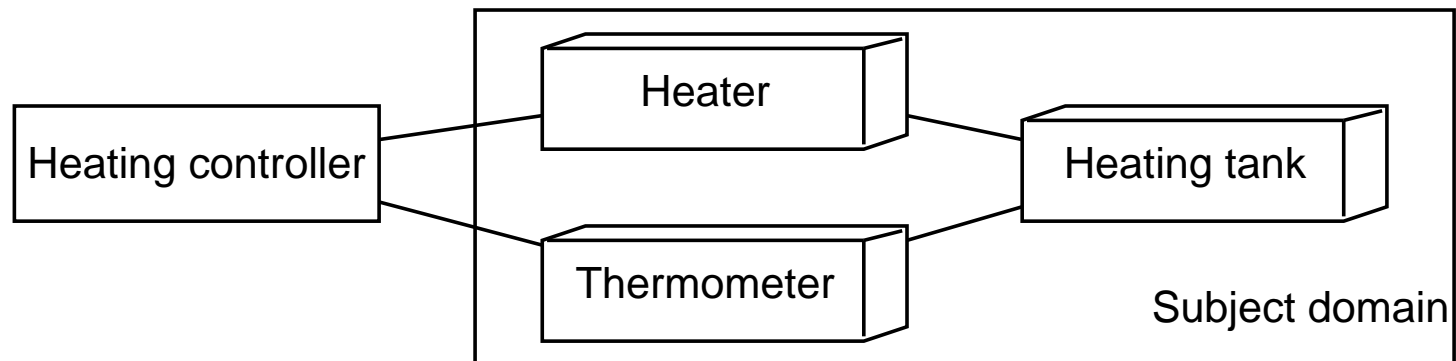
Subject domain

The part of the world talked *about* by the messages that cross the system interface.

To find it, ask what the messages entering and leaving the system are *about*.

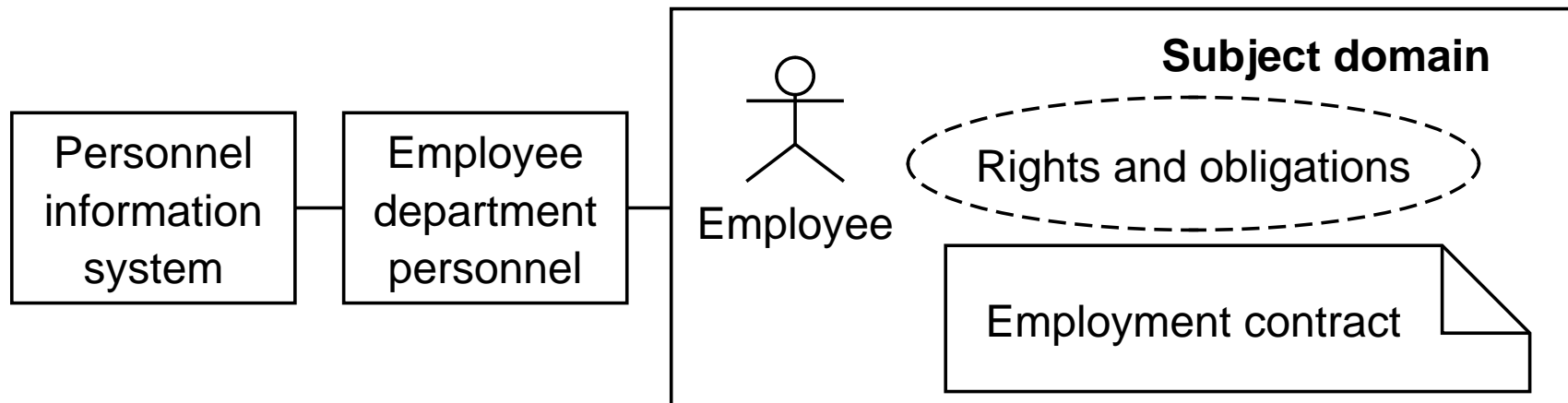
- **Physical entities.** Have a weight and a size. Make noise, generate heat.
 - People
 - Devices
- **Conceptual entities.** Invisible and weightless. E.g. bank accounts, obligations, permissions.
- **Lexical items.** Physical entities with a meaning. E.g. Tickets, contracts, receipts.

Example of a physical subject domain



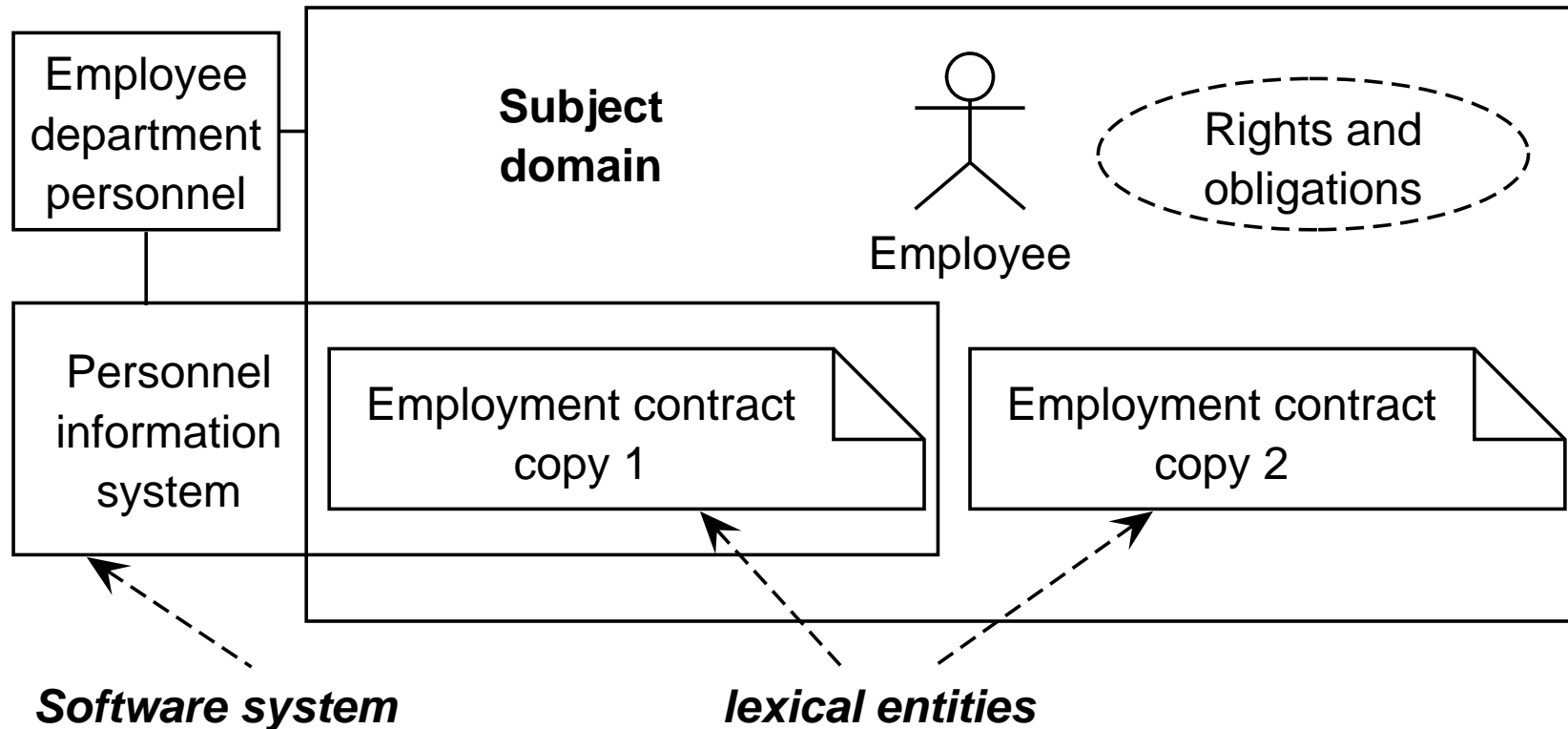
A physical subject domain always exists outside the software system.

Example of conceptual, lexical, physical subject domain entities



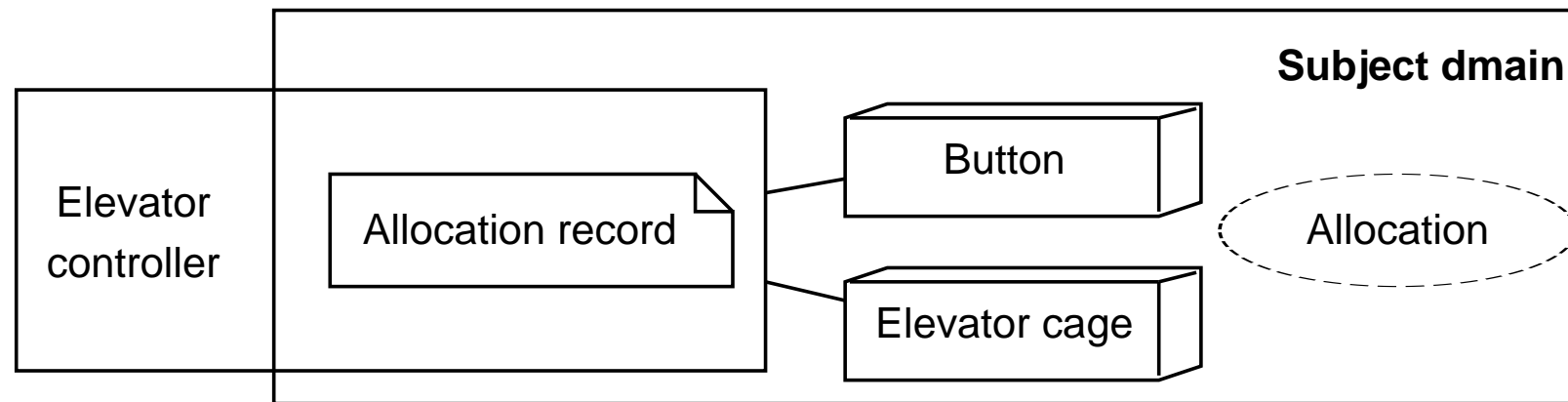
Conceptual entities always exist outside the software system. They exist because people agree to treat them as existing.

Lexical entities can be copied



Lexical subject domain entities may exist in the software or in the environment of the software. They may be copies of each other.

Physical and conceptual entities in a subject domain



Functions of reactive systems

- **Registration.** System registers events in environment.
 - TIS registers participation in course.
 - Heating controller registers temperature.
- **Direction.** System influences events in environment.
 - Controller switches heater on.
 - Compact dynamic bus station controller: Tells driver at which platform to park.
- **Manipulation.** System manipulates virtual entities; this has a meaning in the social environment.
 - ETS stamps ticket.
 - TIS allocates joiner to group.

Connection domain

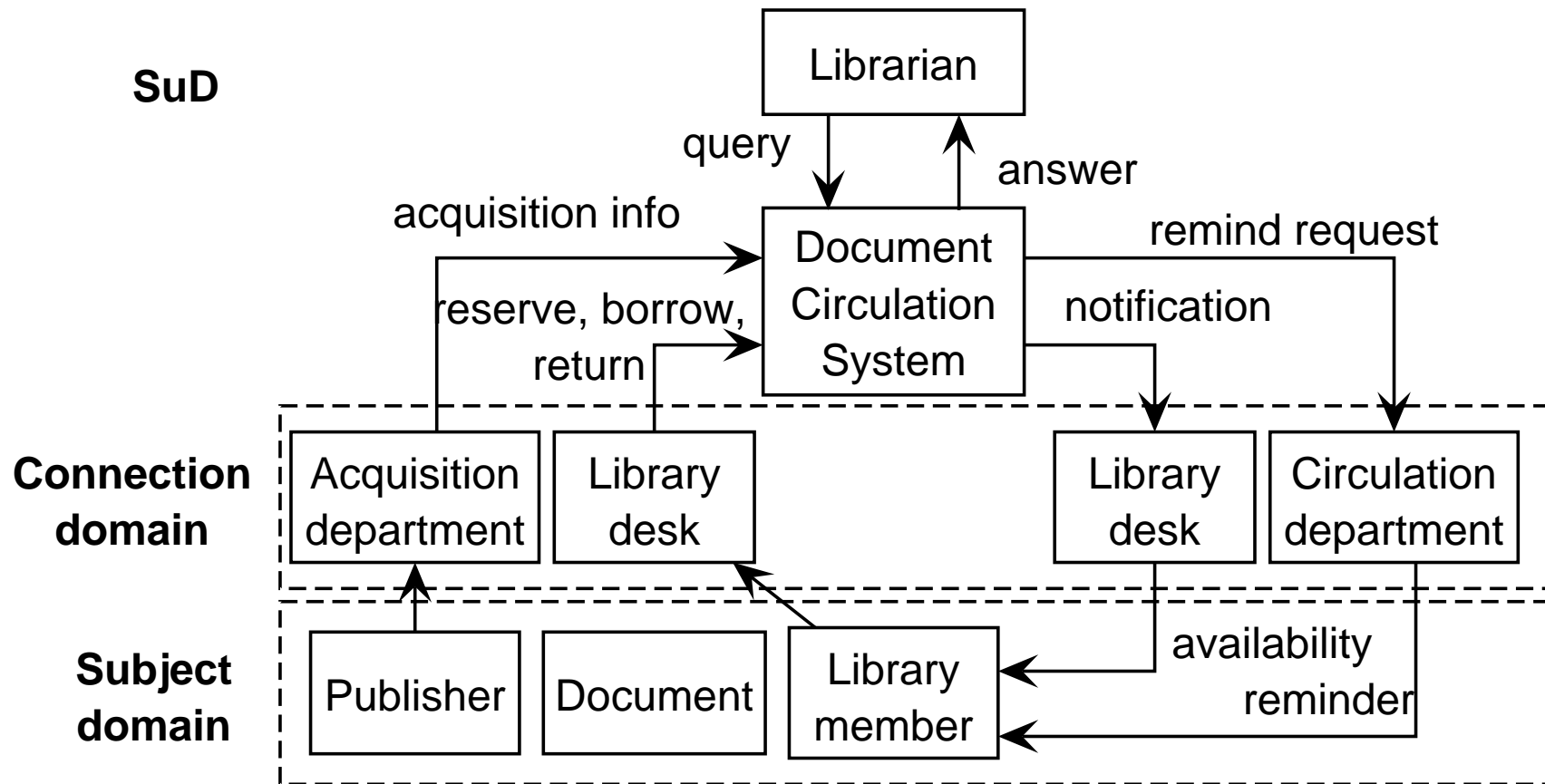
- Events of interest usually do not occur *exactly* at the interface of the system.
- Actions caused by the system usually do not occur *exactly* at the interface of the system.

Communication channel needed. If represented explicitly it is called a **connection domain**. Is possible source of

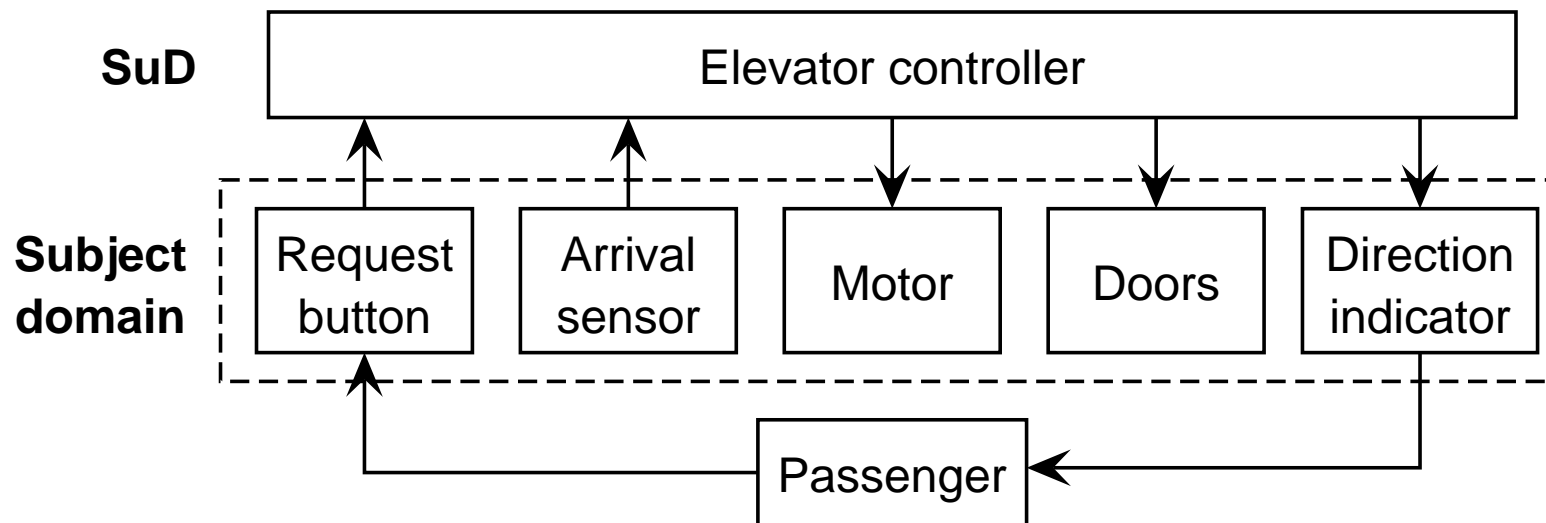
- delay,
- loss, and
- distortion

of messages to/from the system.

Subject domain and connection domain



System directly connected to subject domain



Main Points

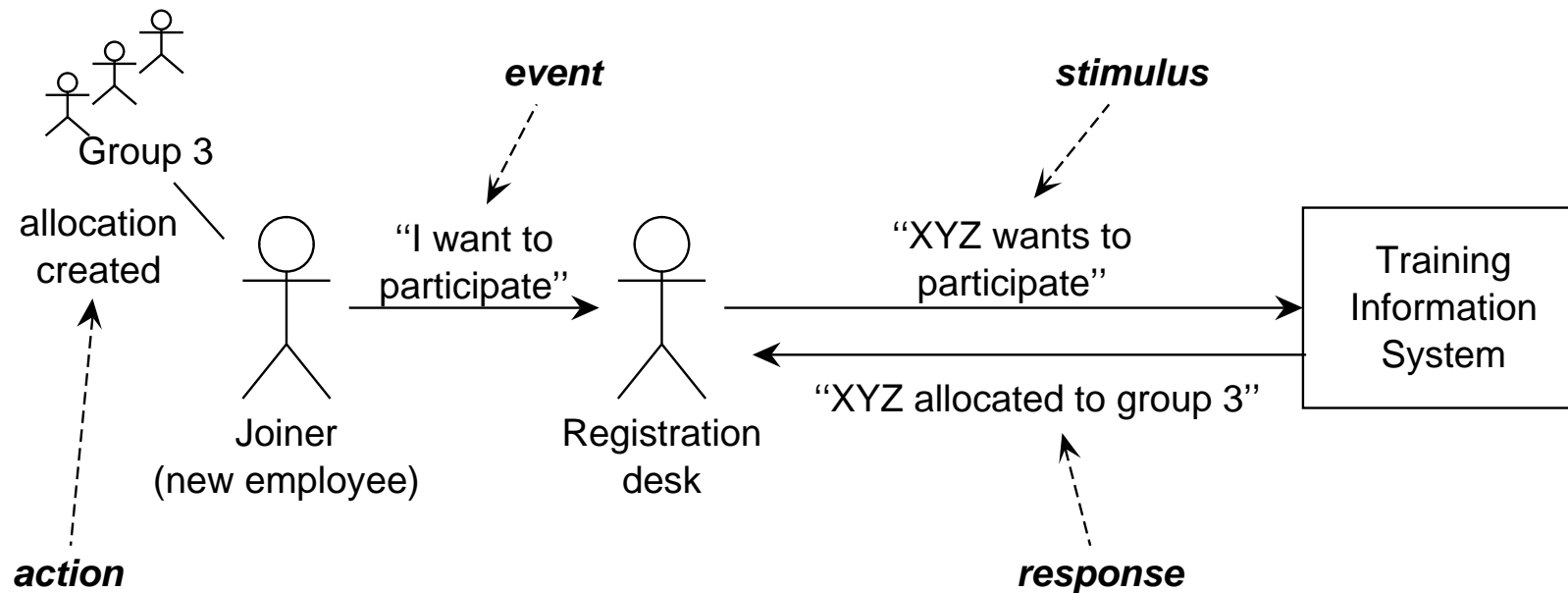
Let SuD be a Software System under Design.

- Message that cross the interface of the SuD are *about* the subject domain.
- Subject domain entities exist *outside* the SuD; except lexical items, that may also exist inside the SuD.
- Messages that cross the interface have three kinds of functions: *Informative, Manipulative, Directive*.
- Messages travel to and from the SuD through a *connection domain*.

Chapter 3. Stimulus-Response Behavior

- A reactive system receives messages from sources in its environment and it sends messages to destinations in its environment.
- At the sources, events occur.
- At the destinations, actions occur.
- We now look at the chain of cause and effect from event to system stimulus, and from system response to action.
- We will see that to interpret stimuli and to motivate responses, we need to make assumptions about the environment.

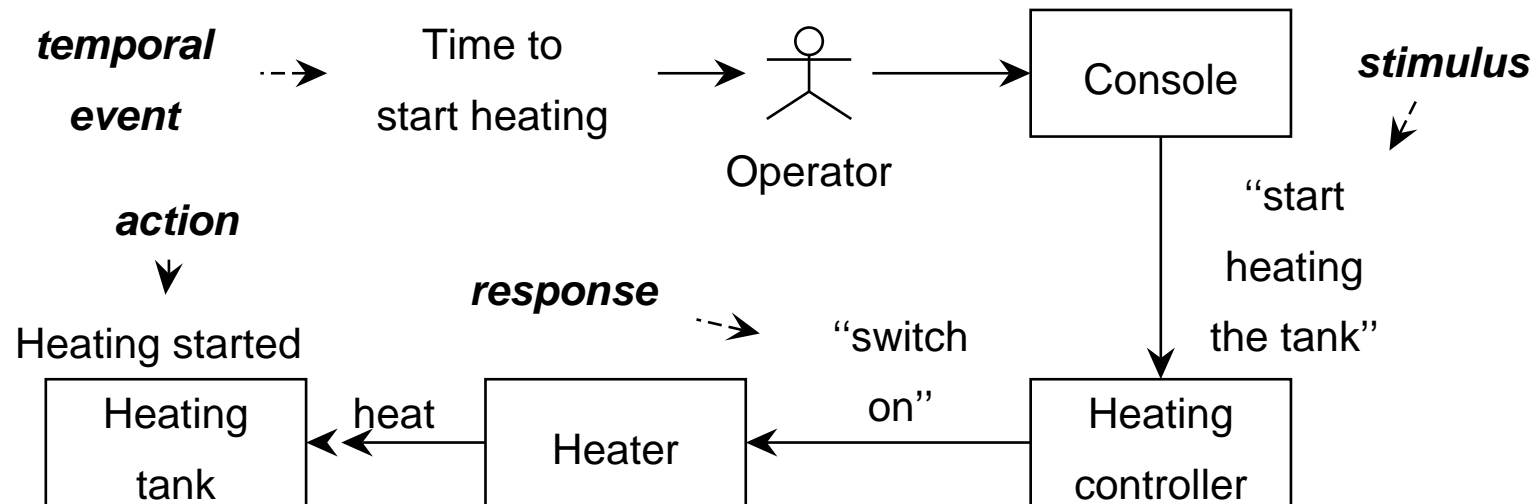
Example 1



Assumptions:

- Registration desk transmits message faithfully.
- The employee is indeed a joiner.

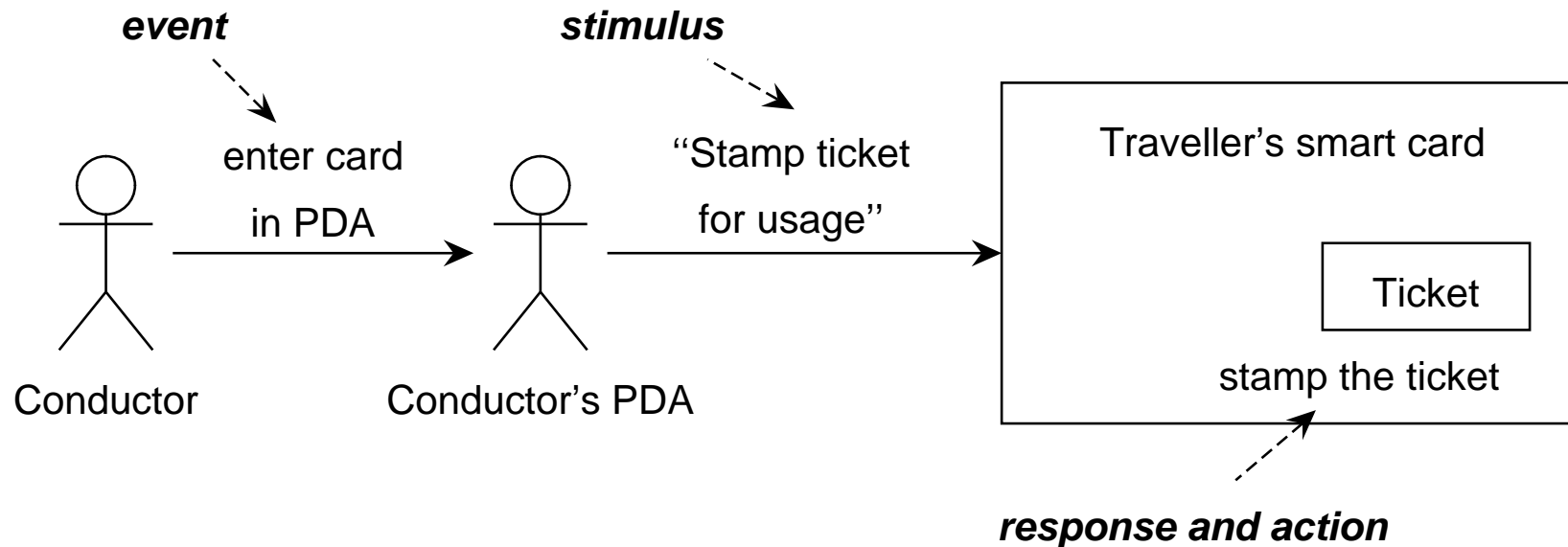
Example 2



Assumptions:

- The message is transmitted faithfully by the console.
- The translates “switch on” into heat production.
- The heater is connected to the heating tank.

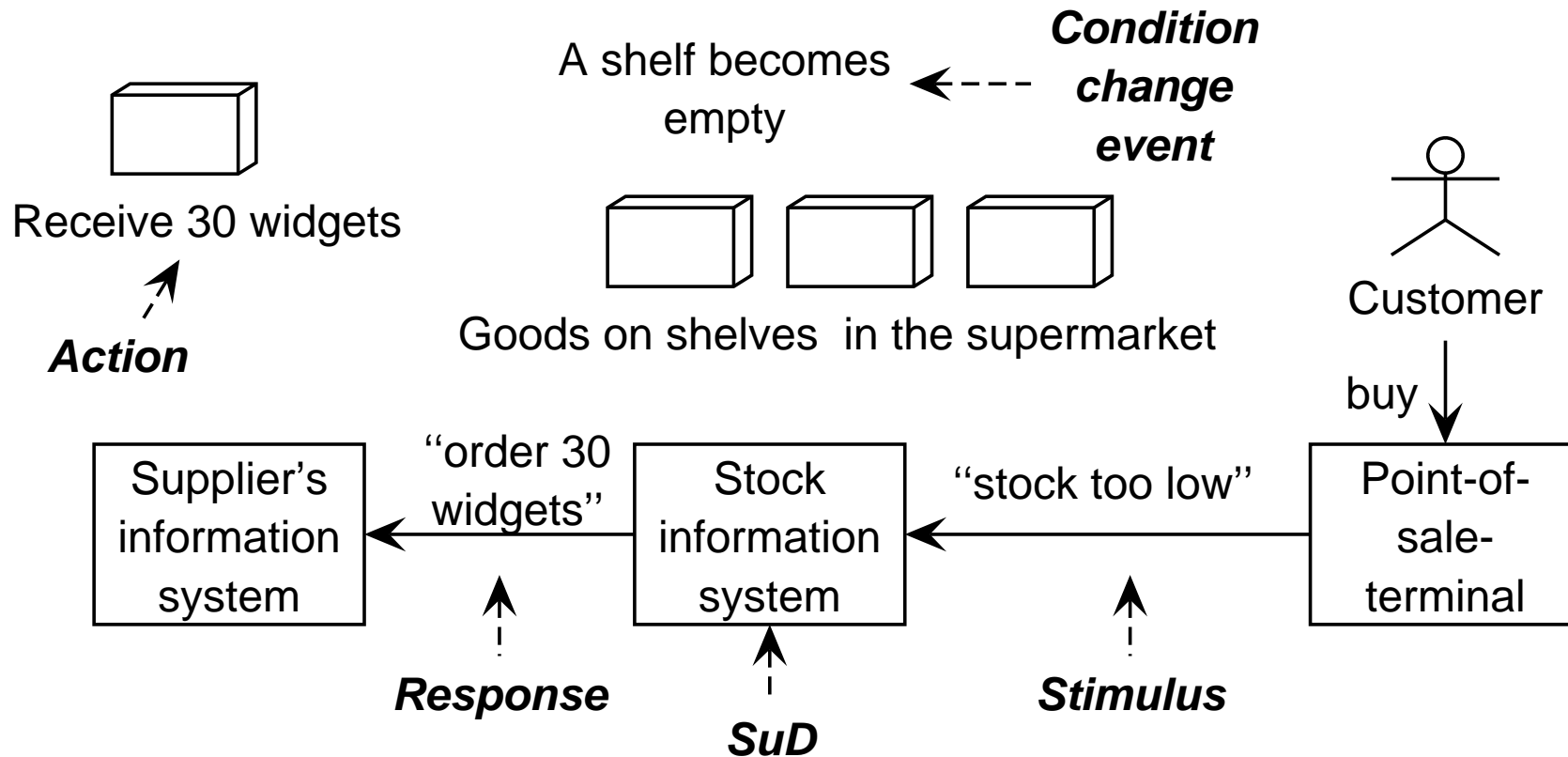
Example 3



Assumptions:

- The PDA is indeed the conductor's PDA.
- The conductor requests to stamp the ticket while the ticket owner is traveling on a segment for which the ticket is valid.

Example 4



SuD discovers the change event by testing the condition $\text{stock} < 4$.

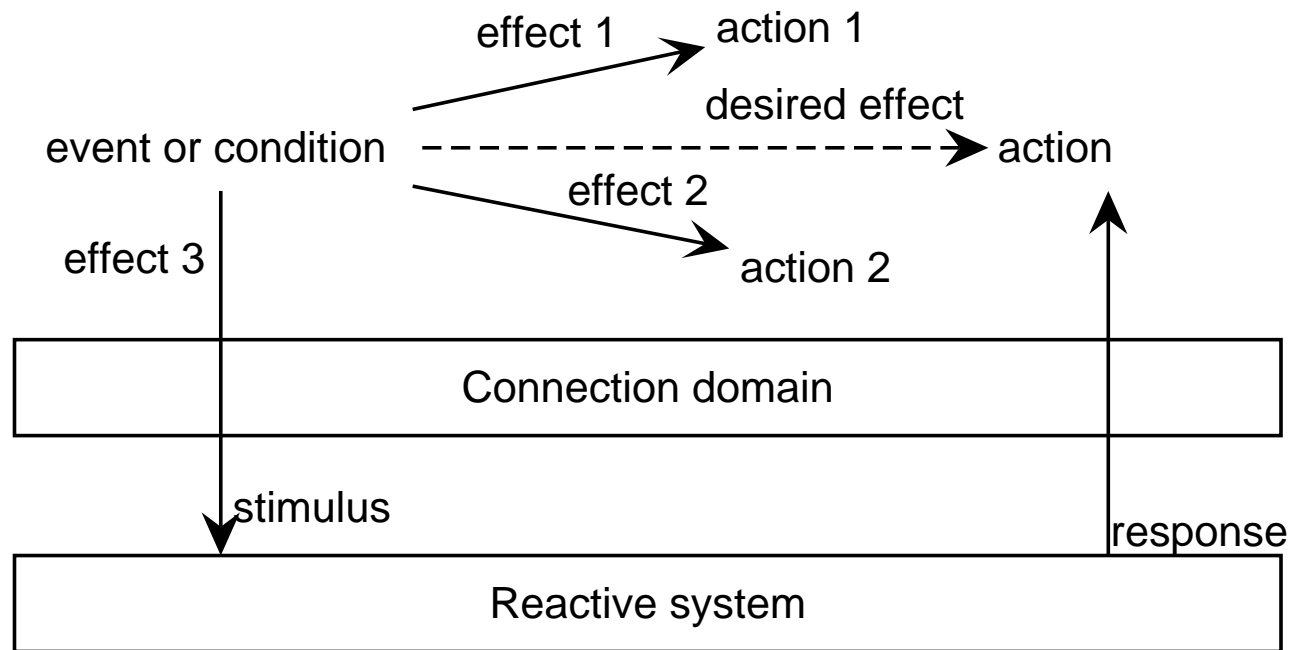
Assumptions:

- Data in SIS accurately represent state of the shelves.

Summary of stimulus-response behavior

- Some *named external event, condition change event* or *temporal event* occurs in the environment.
- Through some communication channel, this causes an SuD *stimulus*.
 - May be modeled explicitly as connection domain.
- The SuD *responds*.
- Through some communication channel, this causes an *action* in the environment.
 - For virtual entities, response and action coincide.

Structure of stimulus-response behavior



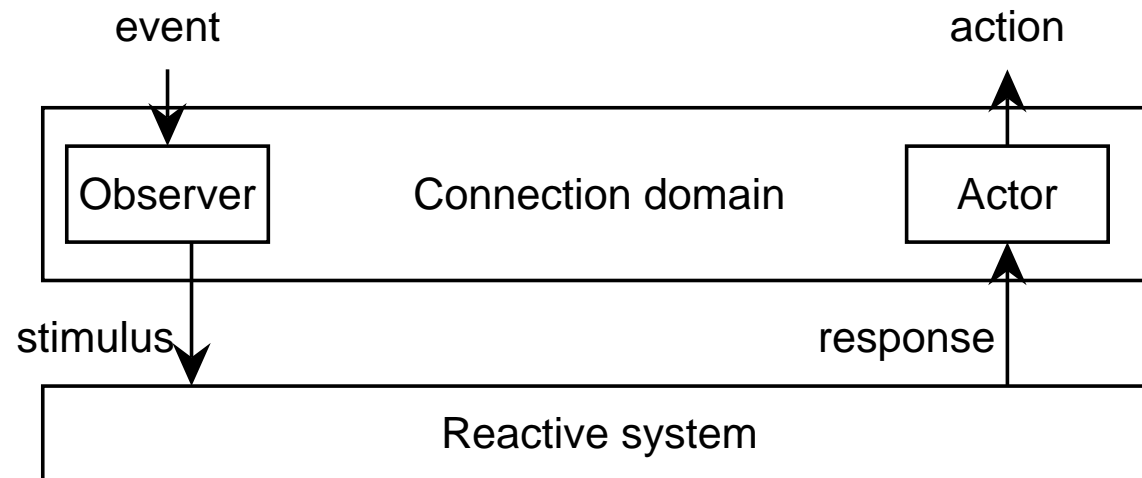
The role of assumptions

These are statements about the environment ...

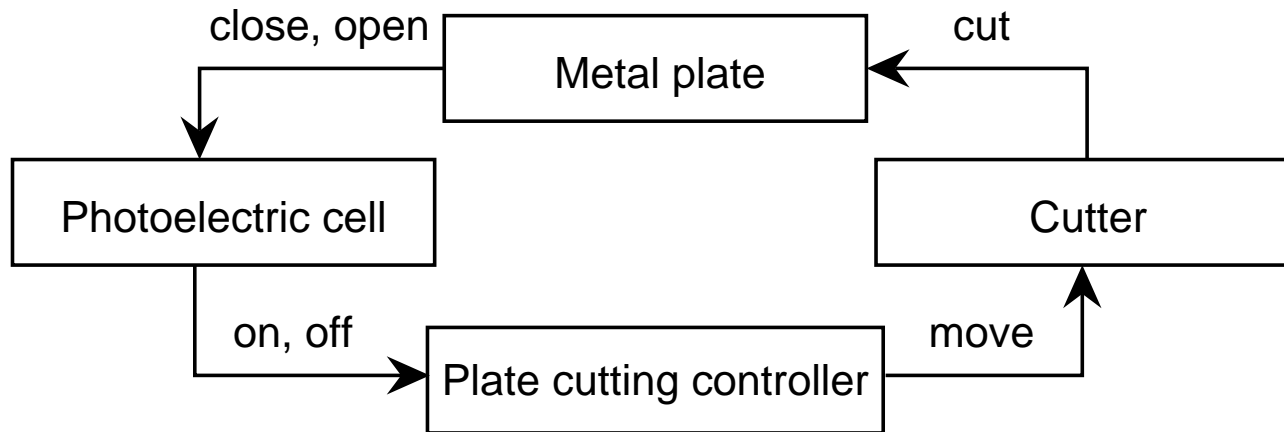
- that must be true for the (stimulus, response) pair to be desirable;
- but the SuD cannot guarantee them to be true.
- If assumption is false, (stimulus, response) pair may still be desirable, or indifferent, or even undesirable!

Typical assumptions concern laws of nature and properties of devices (e.g. observers and actors) and people (e.g. users and operators).

Observers and actors are in the connection domain



Assumptions about observers



- Events of interest: begin of plate arrives, end of plate arrives
- Available stimuli: on, off.

We need the following assumptions:

- The photo-electric cell is functioning properly.
- The cell is stimulated only by the arrival of the begin or end of a metal plate.

Event recognition

System must respond to the event

- a point arrives where the sheet must be cut.

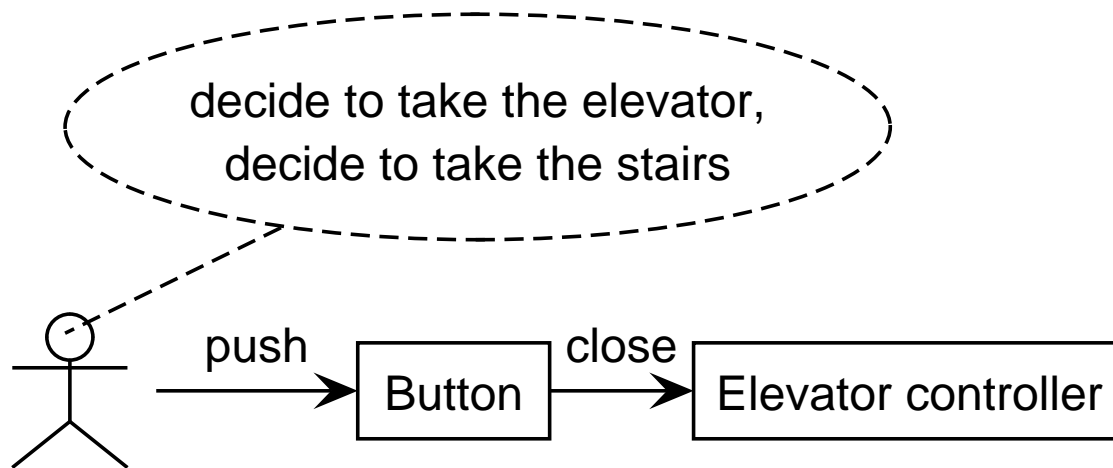
The system must infer the occurrence of this event. **Event recognition:**

- Record the point in time at which plate arrives.
- Wait for time at which desired point is under the cutter.

Additional assumption needed:

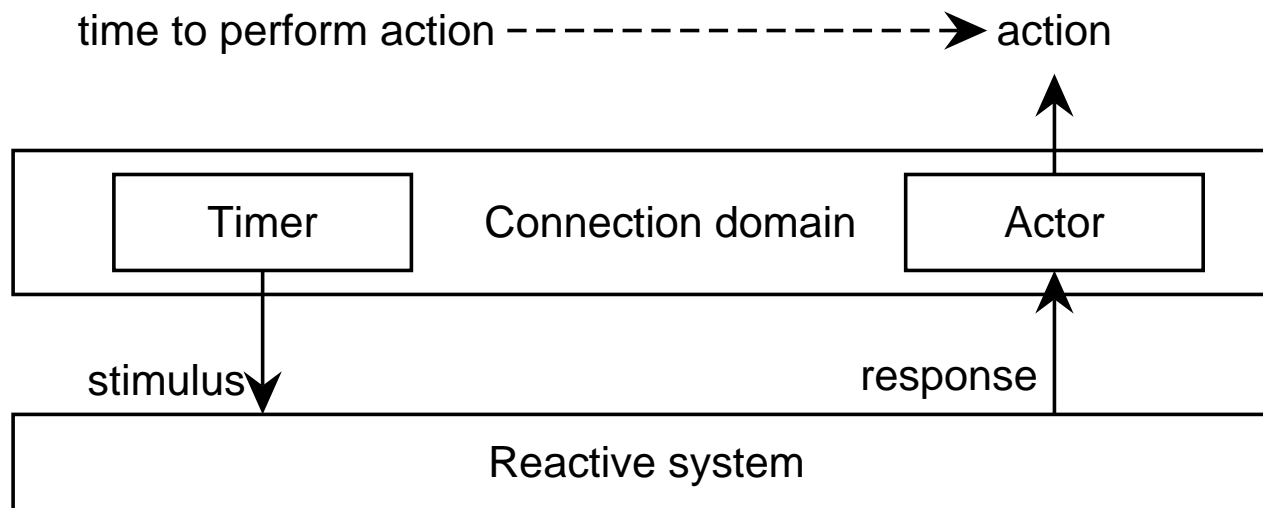
- Speed of the plate is n meters / second.

Observability of events

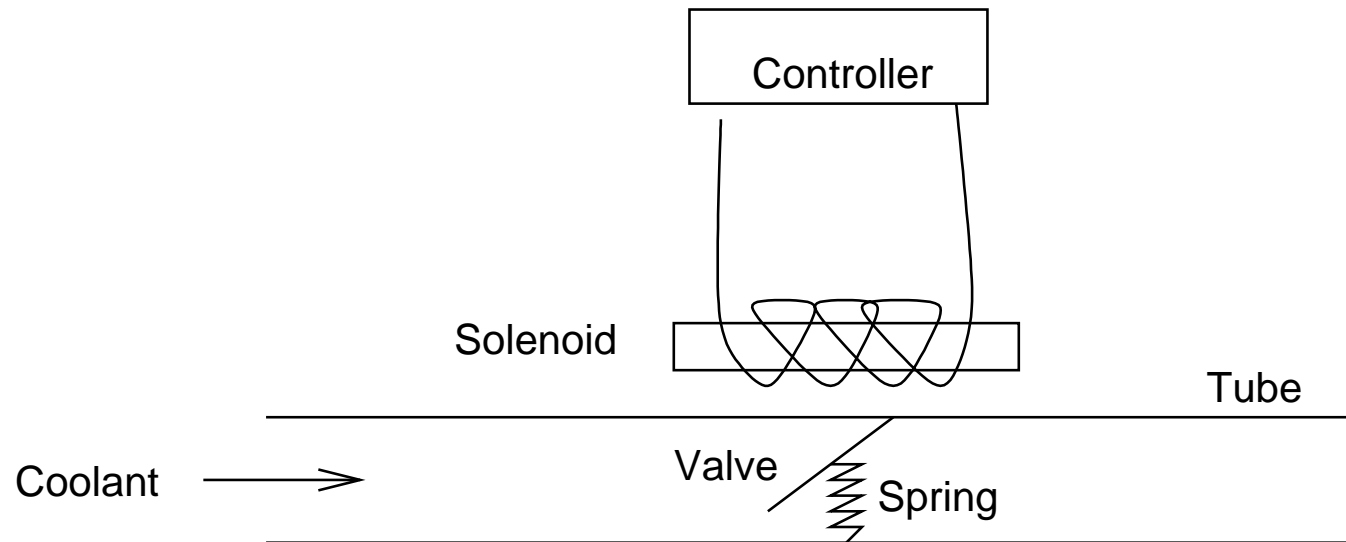


What is the subject domain of the elevator controller?

Observing temporal events



Realizability of actions



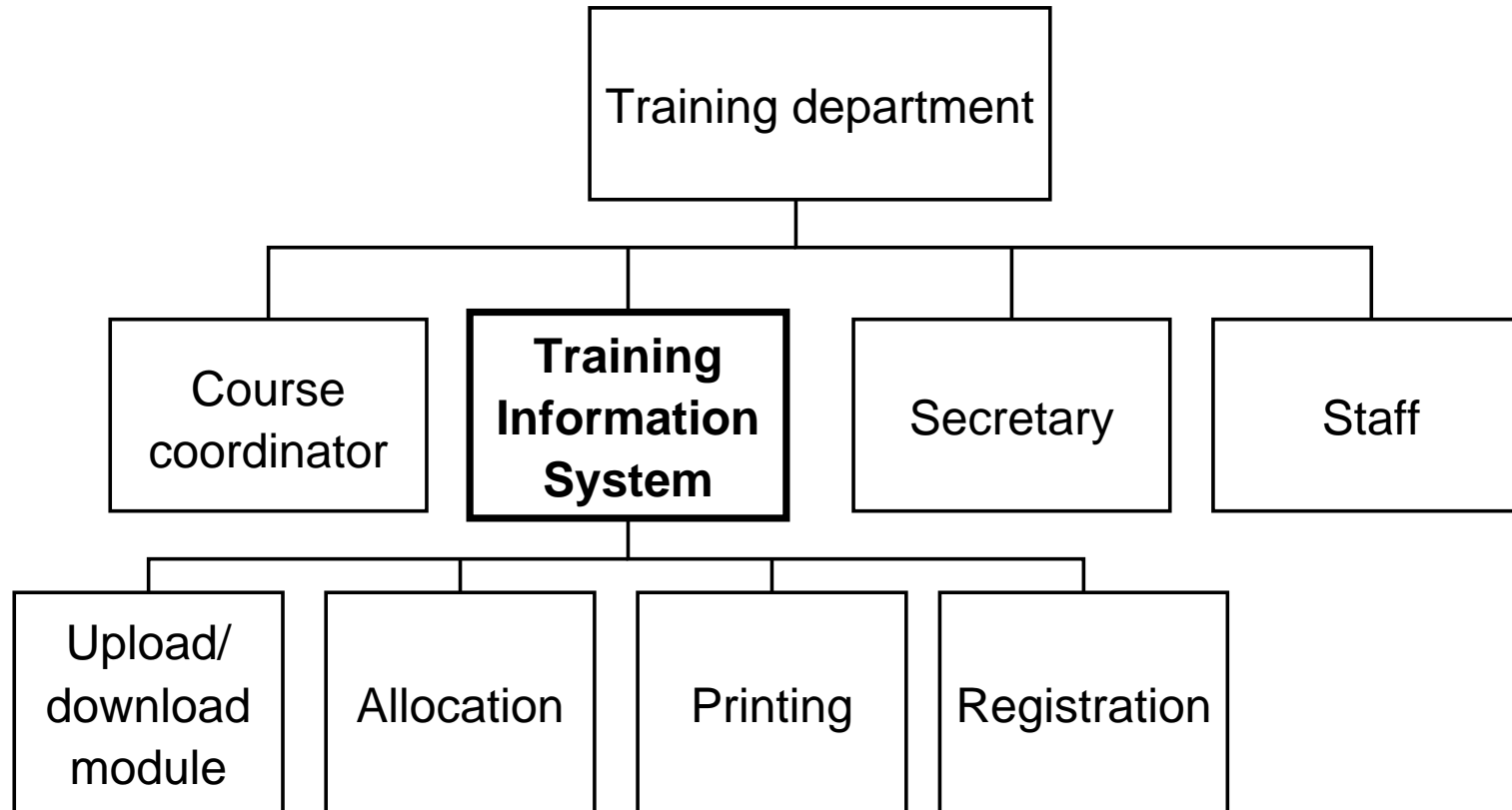
Main points

- Reactive software system is connected to interesting events and actions by a communication channel.
- Stimulus is event observation.
- Response is assumed to cause desired action.
- If assumptions about environment are true, (stimulus, response) pairs are desirable; otherwise, they may be undesirable or indifferent.
- Assumptions cannot be guaranteed by the system.

Chapter 4. Software Specifications

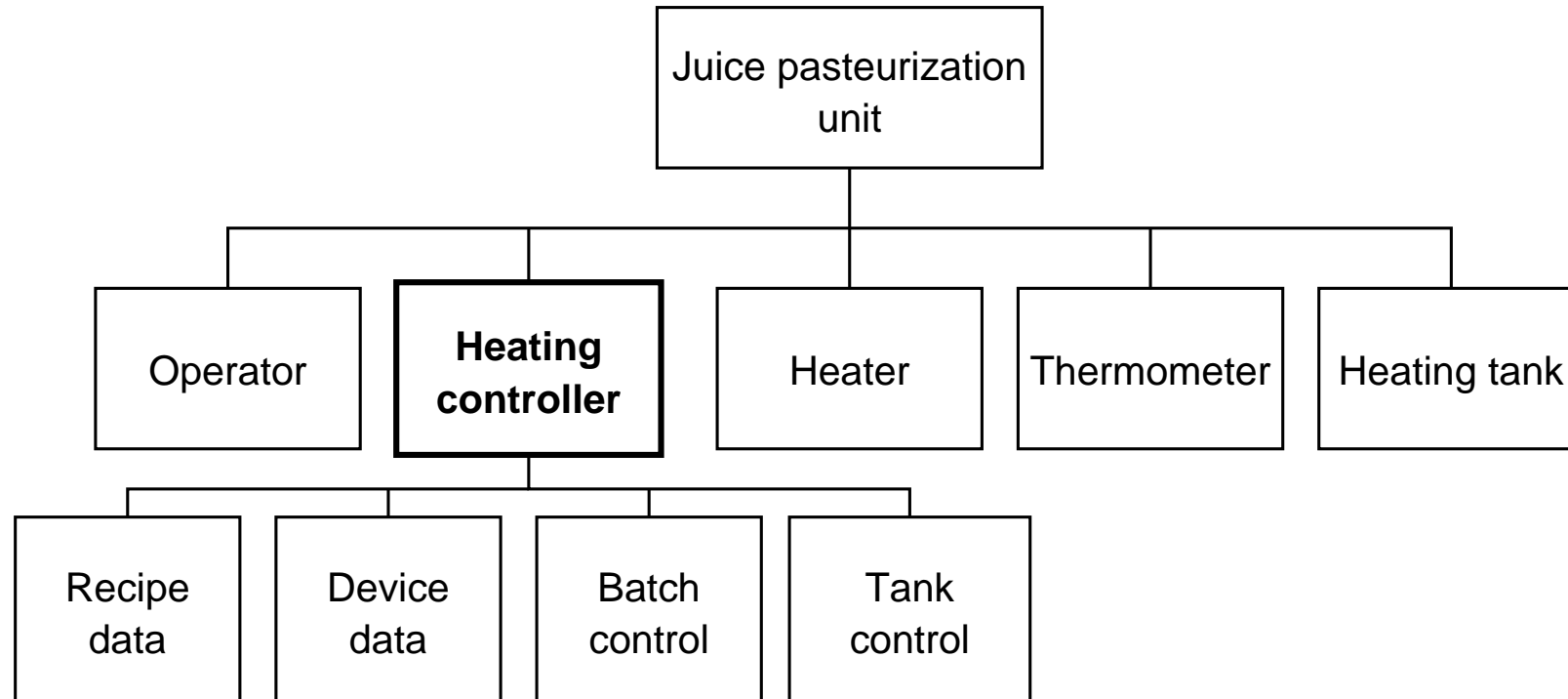
- We call any creative decision about a system a design decision.
- To design a system is to make a plan how it will be built.
- A specification is a description of design decisions.
- A reactive system specification must describe
 - the place of the SuD in the system hierarchy,
 - it must describe functions, behavior and communication of the SuD, and
 - it must describe its composition.
- The specification must be used to motivate the design in a systems engineering argument.

Systems engineering argument example 1



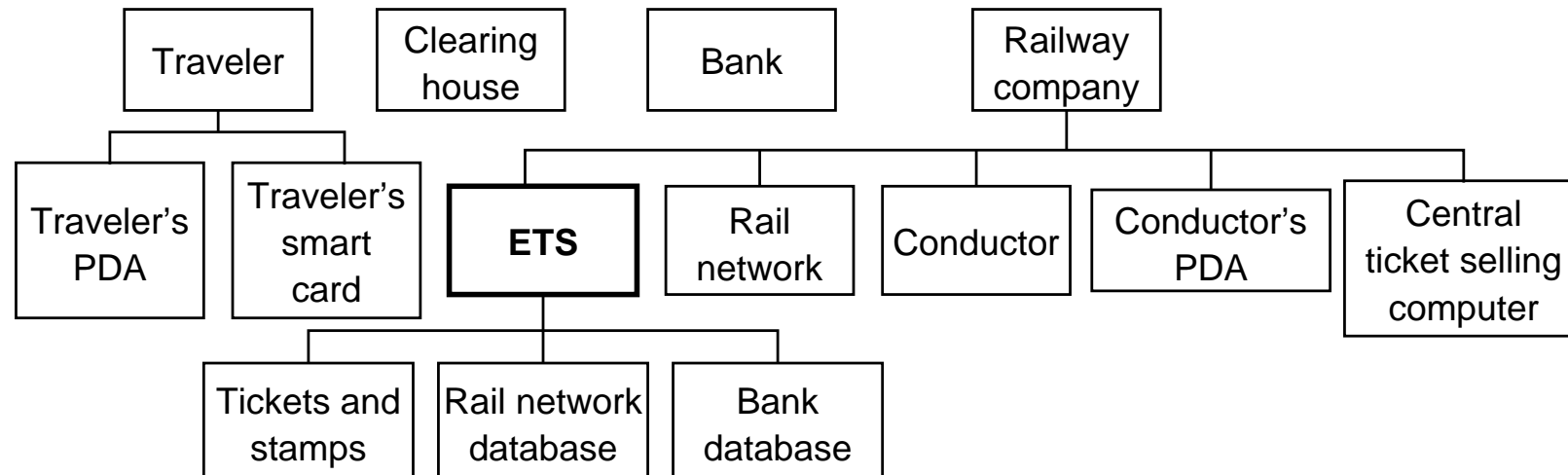
- If TIS allows registration of unexpected participants
- and the department keeps extra staff,
- then department is able to handle newcomers efficiently.

Systems engineering argument example 2



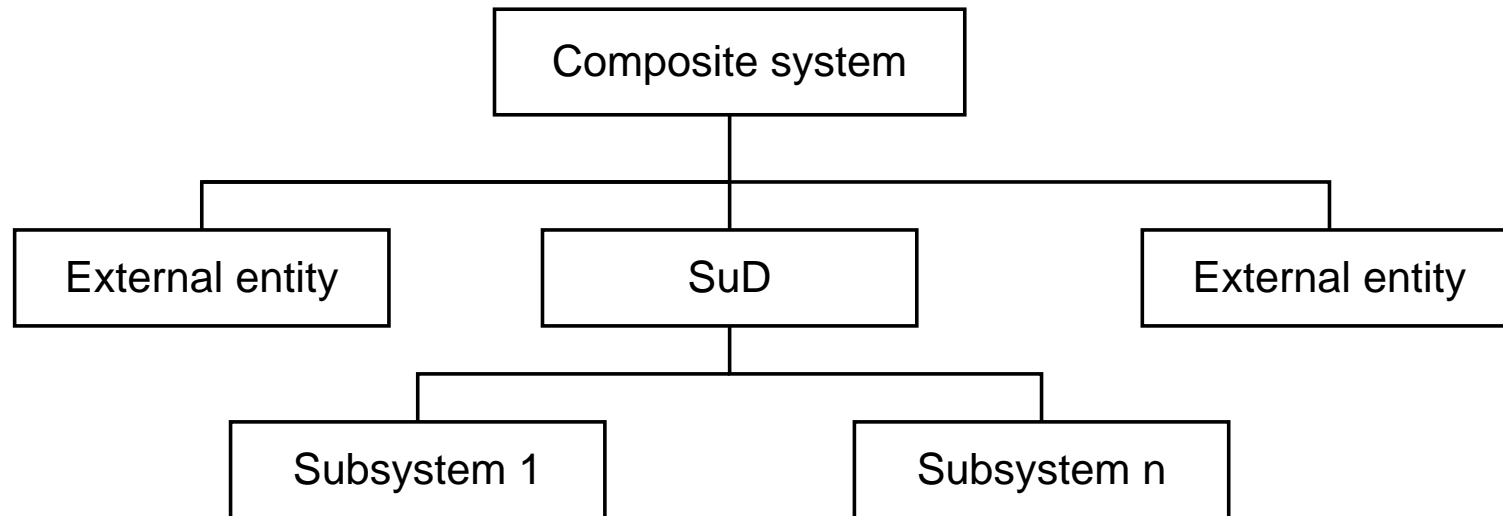
- If controller controls heater according to batch recipe
- and thermometer is functioning,
- then pasteurization unit heats batch according to recipe.

Systems engineering argument example 3



- If ETS allows travelers to buy tickets through their PDA
- and traveler's PDA interfaces with ETS,
- then railway company reduces operating costs.

System engineering argument



- If SuD satisfies specification S
- and environment satisfies assumptions A
- then composite system has emergent properties E .

Emergent properties arise by interaction of component systems.
They should satisfy *goals* of composite system.

The role of assumptions

- If assumptions are not satisfied by environment, the composite system goal may not be reached.
- System cannot guarantee the assumptions.

Examples:

- Heat will rise when “switch on” sent to heater.

Assume laws of thermodynamics and assume that devices work.

- Ticket is stamped for segment of the current route.

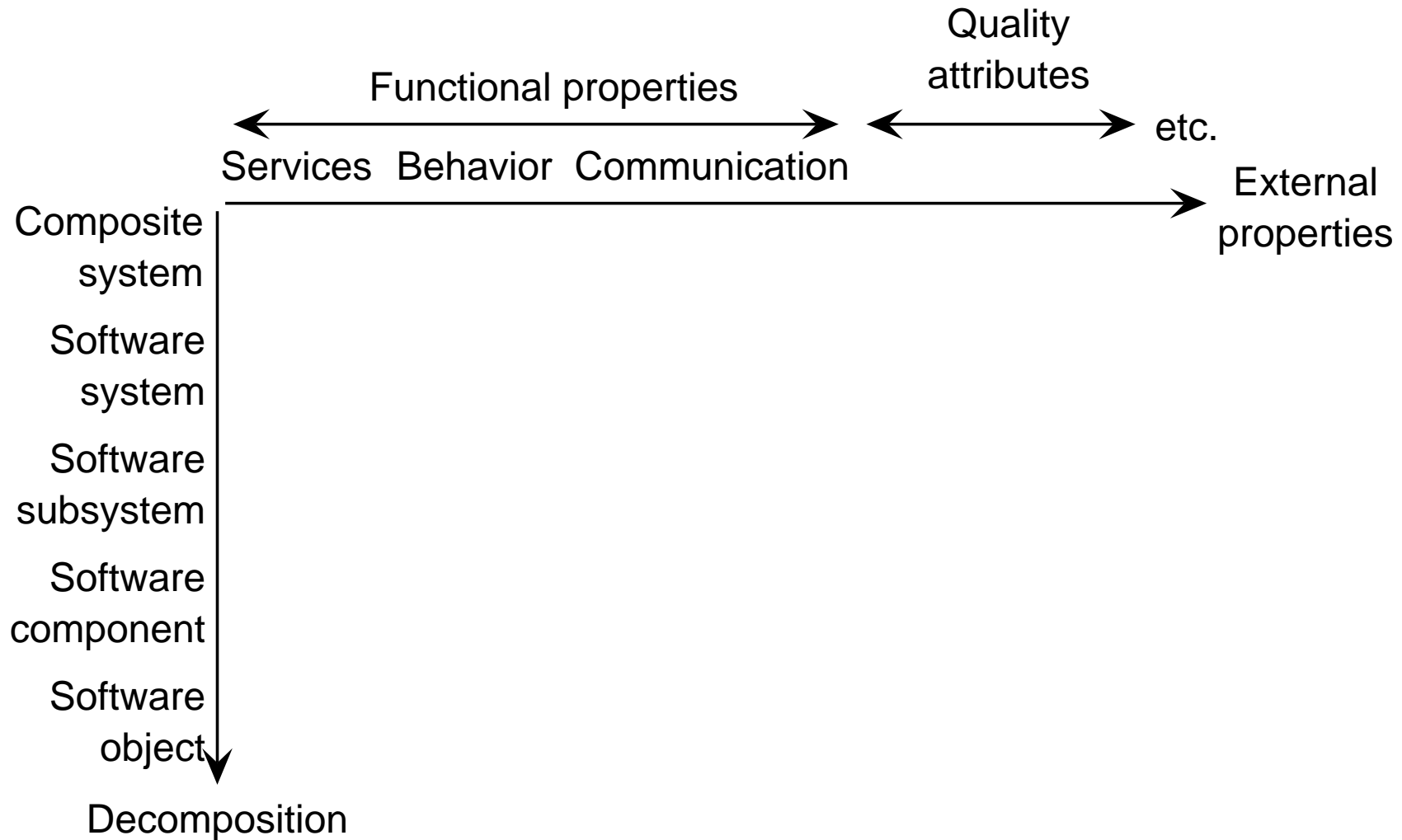
Assume that the conductor, traveler and smart card are physically located in the segment for which ticket is stamped.

Kinds of assumptions

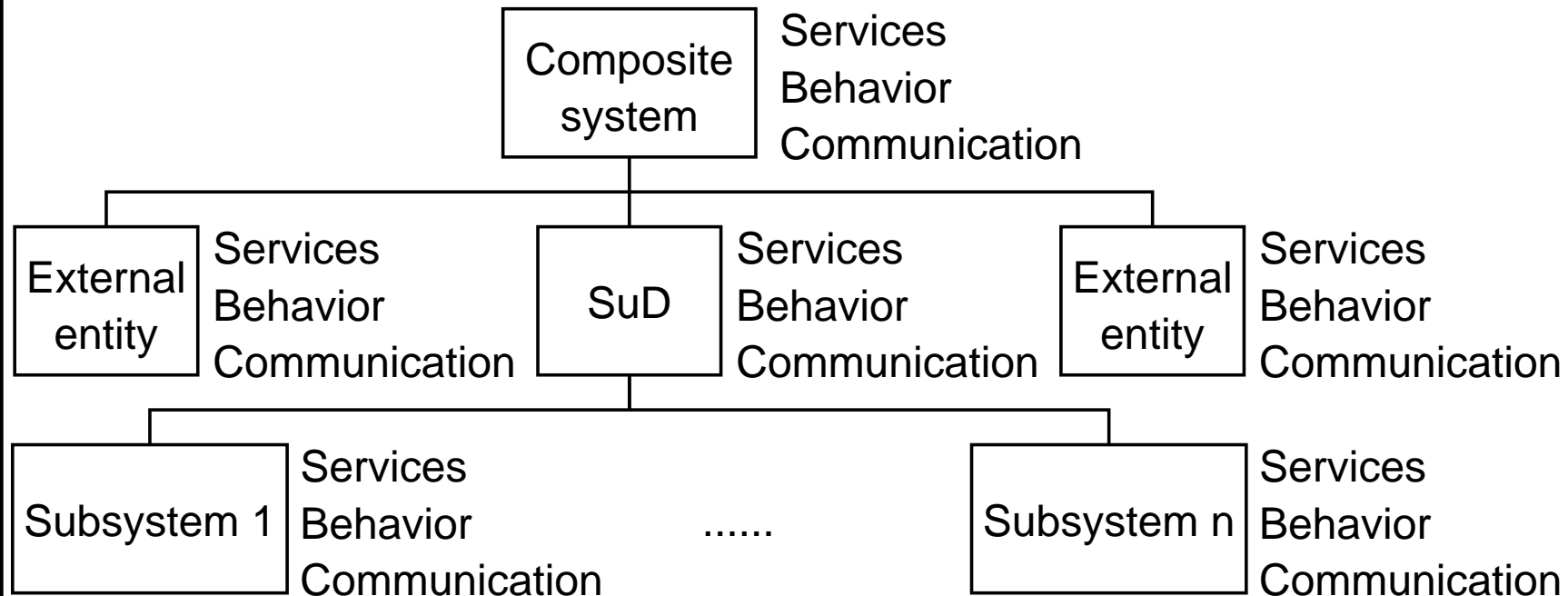
- laws of nature
- specifications of devices
- rules for people (laws, procedures)
- definitions of conceptual structures (e.g. meaning of stamps and tickets)

Only laws of nature are infallible ... we assume. Many assumptions are about the subject domain and about the connection domain.

Kinds of properties



Properties occur at every level



Functional properties appear at every level.

- Service = Interaction that delivers desired effect.
- Behavior = Ordering of interactions over time.
- Communication = Symbol flow between different entities.

Terminology

With respect to the SuD, we talk about:

- Requirement = desired property.
- Constraint = imposed property.
- Aspect = group of properties.

Operational specification

Specification of set of reproducible operations to find out whether a property is present.

Important class of operational specs has the form

- If stimulus s occurs
- and system is in state C
- then produce response r .

Also called Event-Condition-Action (ECA) rule. Appears in state transition table (see chapter 12).

Main points

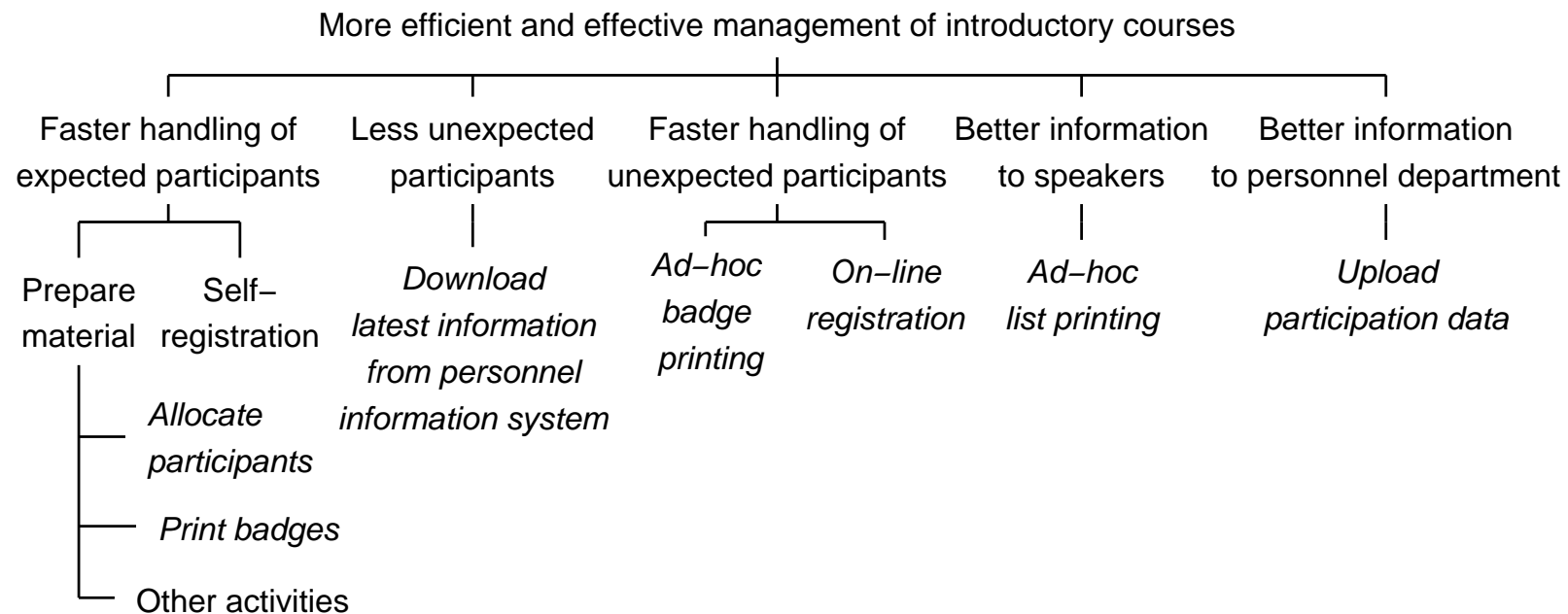
- Systems have functional properties and quality properties.
- Functional properties are services, behavior, communication.
- These reappear at every level in aggregation hierarchy.
- (There are really three aggregation hierarchies.)
- SuD needs external entities to jointly produce desired emergent properties of composite system. System engineering argument.

Chapters 5–7. Mission Statement, Function Refinement Tree, Service Description

- We describe the utility of the system for its environment by describing its functions for the environment.
- But a function is not a component; it is not a program; it is not a part of a program. It is the added value, or utility, of the SuD for its environment.
 - “The function of this coffee machine is to brew coffee.”
 - “The added value of this coffee machine is to brew coffee.”
 - “The utility of this coffee machine is to brew coffee.”

In this course, these three sentences mean the same thing.

Example 1: Goal tree of the training department

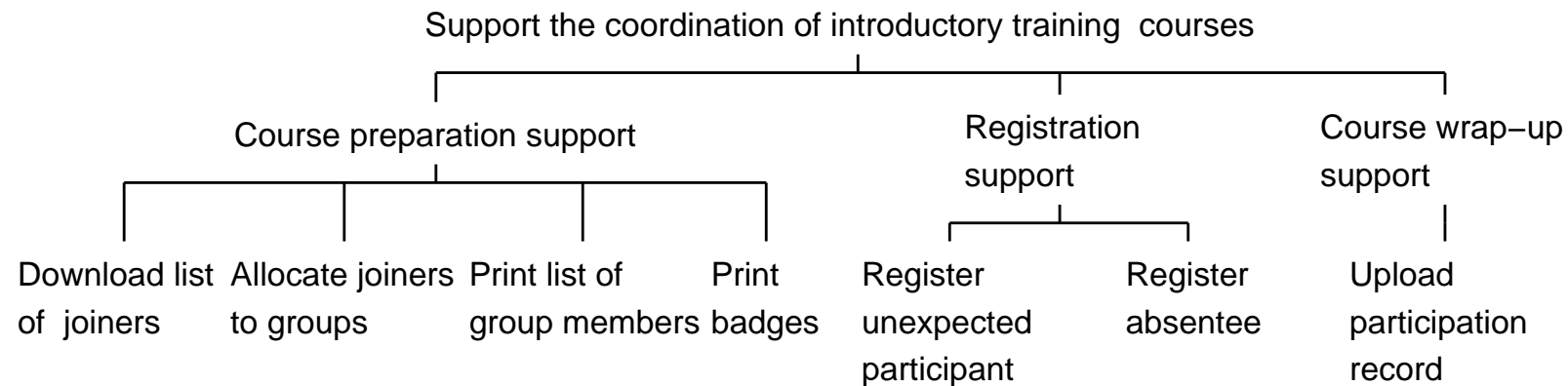


TIS should contribute to goals in italics.

Example 1: Mission of TIS

- **Name:** Training Information System
- **Acronym:** TIS
- **Purpose:** To support the management of monthly introductory training courses.
- **Responsibilities:**
 - To support course preparation, including allocation of participants to groups and printing badges
 - To support course handling (unexpected, absentees)
 - To support course wrap-up
- **Exclusions:**
 - Data of unexpected participants is not checked.
 - No support for allocation of speakers to groups.
 - No support for allocation of groups to rooms.

Example 1: Function refinement tree of TIS



Example 1: Description of a TIS service

Download joiners

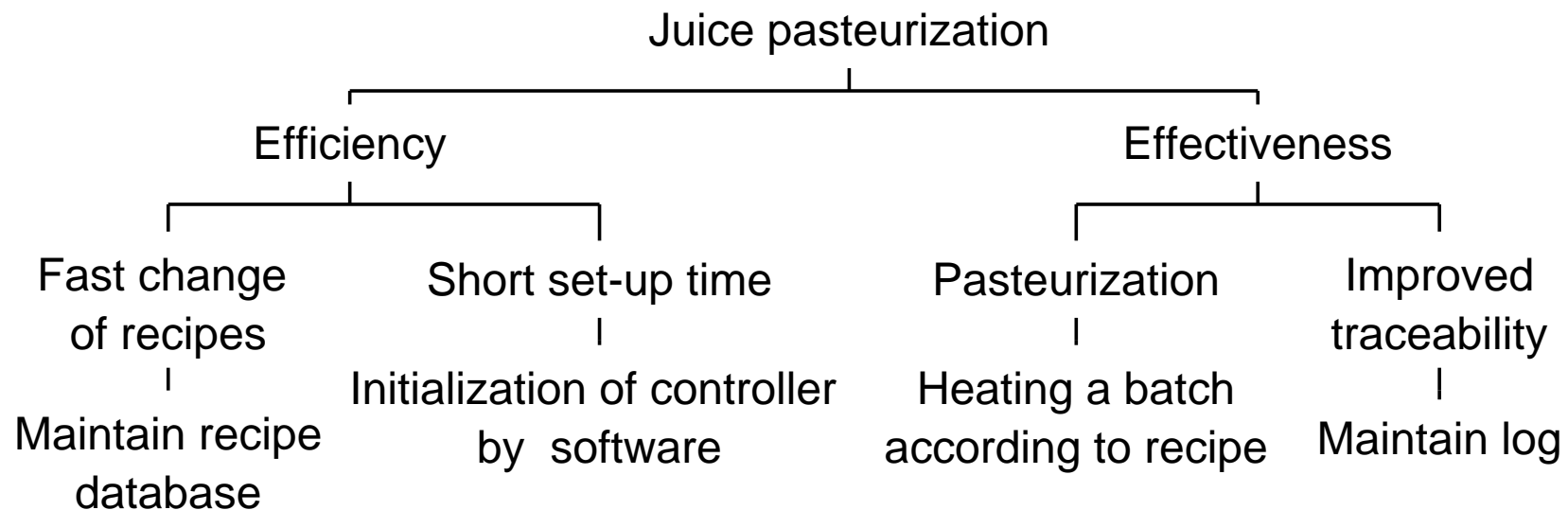
- **Triggering event:** Coordinator requests to download list of joiners from the personnel information system.
- **Delivered service:** Download the list of people from the Personnel Information System who have joined the company since the previous training.
- **Assumptions:** The data in the Personnel Information System reflects the situation accurately with a time lag of not more than one working day.

Example 1: Description of another TIS service

Upload participant record

- **Triggering event:** Coordinator requests to upload list of joiners to personnel information system.
- **Delivered service:** Upload the list of people who participated in the training to the Personnel Information System.
- **Assumptions:** The Personnel Information Systems is able to deal with data about unexpected participants, including any remaining errors in that data.

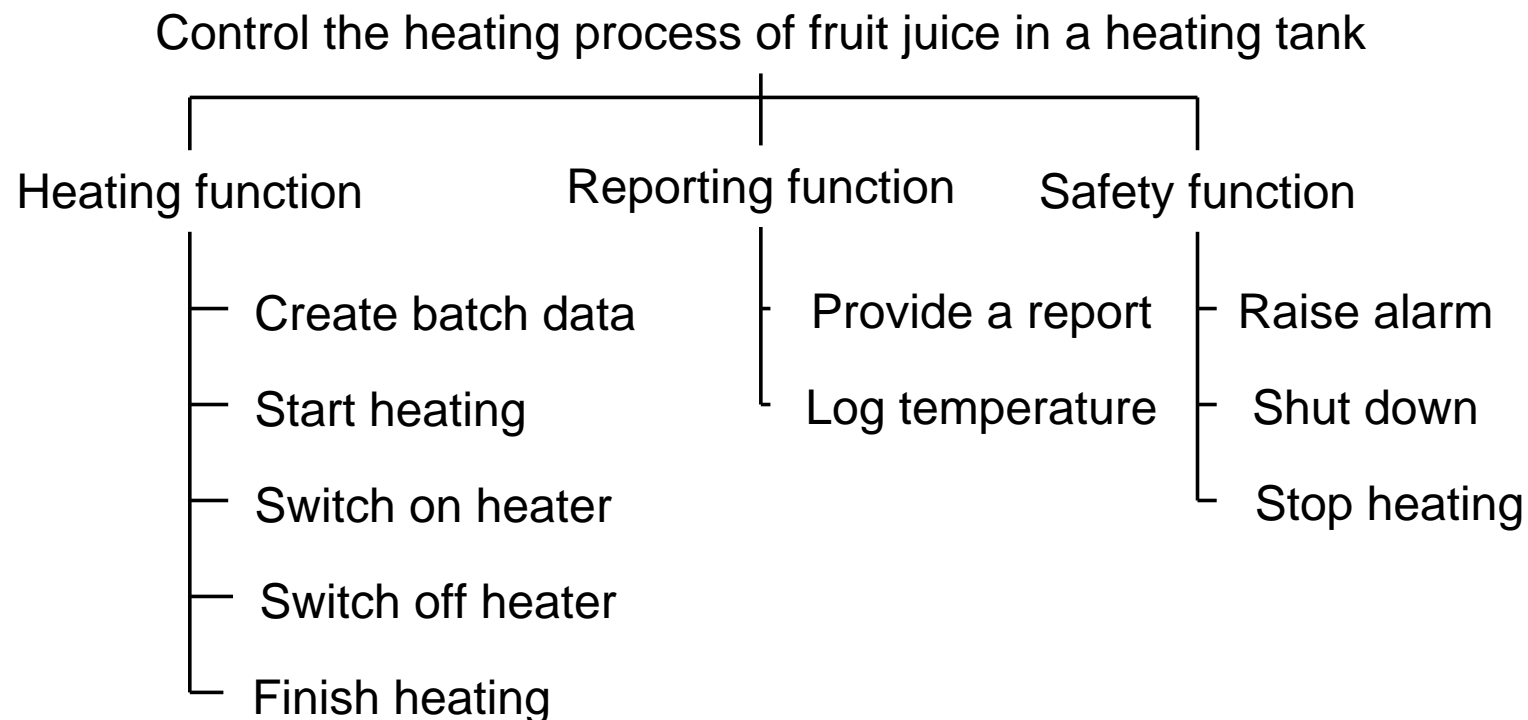
Example 2: Goal tree of juice pasteurization plant



Example 2: Mission of heating controller

- **Name:** Juice heating controller.
- **Purpose:** To control the heating process of fruit juices in a heating tank.
- **Responsibilities:**
 - To initialize itself with batch data and heat the batch according to recipe.
 - To report on the heating process.
 - To maintain safe conditions in a tank.
- **Exclusions:**
 - Filling the storage tanks with juice.
 - Transferring pasteurized juice to the canning line.

Example 2: Function refinement of heating controller



Example 2: A service description

- **Name:** P1. Heat batch according to recipe.
- **Triggering event:** Operator command “start heating batch b according to recipe”.
- **Delivered service:** Upon reception of this command, the controller ensures that a heating process takes place in the heating tanks in which b is stored, according to the recipe of b.
- **Assumptions:** There is a batch in the heating tank.

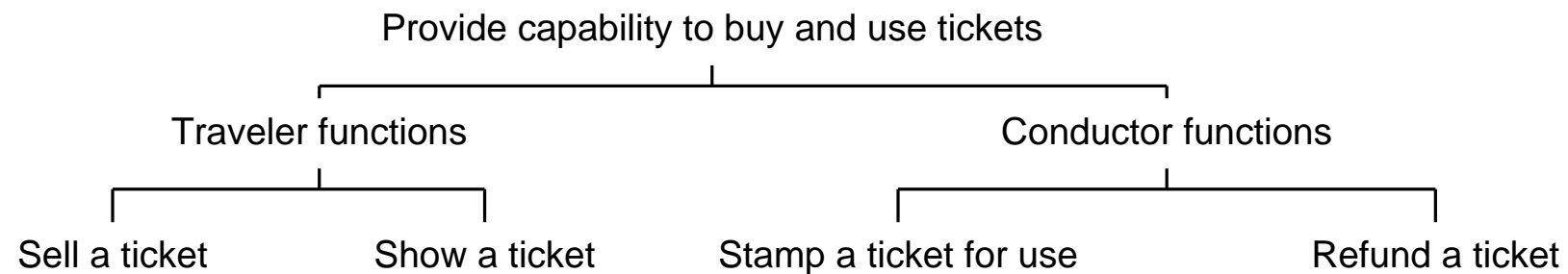
Example 2: Another service description

- **Name:** Log temperature.
- **Triggering event:** When an execution of P1 starts, and then every 10 seconds during this execution of P1.
- **Delivered service:** The controller records the measured temperature in each tank in which b is stored.

Example 3: Mission statement of ETS

- **Name of the system:** Electronic Ticket System (ETS).
- **Purpose:** Provide capability to buy and use tickets of a railway company using a PDA and a smart card.
- **Composition:** Software distributed over smart card, PDA's, central computer.
- **Responsibilities of the system:**
 - To support ticket buying
 - To support ticket usage
 - To support ticket refunding
- **Exclusions:**
 - The system does not perform travel planning
 - Only tickets for one person and one trip (single or return).

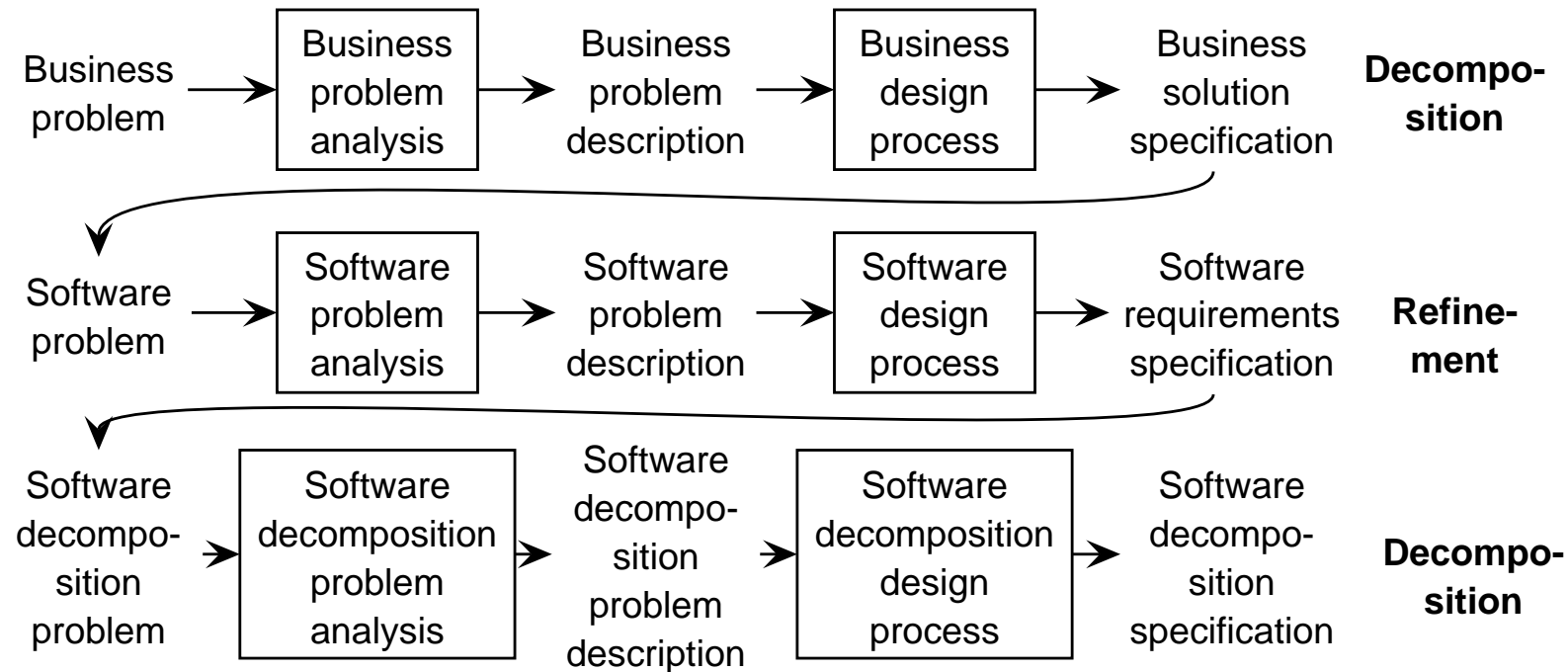
Example 3: Function refinement of ETS



Two service descriptions of ETS

- | |
|--|
| <ul style="list-style-type: none">• Name: Sell a ticket.• Triggering event: Traveler requests to buy a ticket.• Delivered service: Allow a traveler to buy a ticket at any time and place chosen by the traveler. |
| <ul style="list-style-type: none">• Name: Show a ticket.• Triggering event: Traveler requests to view a ticket.• Delivered service: Display ticket attributes to the user. |

Levels of design



- A business solution specification describes a decomposition of the business that solves a business problem.
- A software specification refines the software part of a business solution.

Goal analysis

- First separate the design levels as on previous slides.
- Next identify business goals.
 - In a goal tree, achievement of children goals is sufficient to achieve parent goal.
 - Leaves of a business goal tree usually are desirable business activities.
- From the goals of the business solution, derive from the software goals.
- This gives us statement of purpose and major responsibilities of the software.

Mission statement

Highest level software specification. Talks about software solutions instead of business solutions.

- ✓ To find software mission, analyze business goals.
- ✓ To find mission, find desired emergent properties E of composite system.
- ✓ Justify mission and responsibilities by system engineering argument: Mission statement + environment assumptions entail business goals.

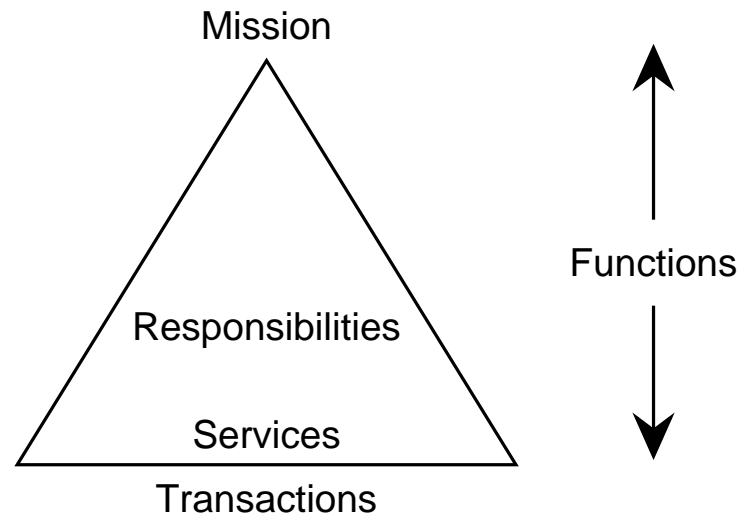
Mission statement is updated as our understanding improves during the project.

Function refinement tree.

Makes responsibilities more specific.

- Not a system structure: Just an indented shopping list.
- Organization of tree is subjective: Determined by discussion with customer.
- The tree bounds functionality of the system.
- It is the most implementation-independent description of the system.
- It justifies the presence of services.
- It prevents the presence of unnecessary services.

Terminology



Function = ability to create desired effect in the environment.

- **Responsibility** = Contribution to environment goal.
- **Service** = Useful interaction triggered by event.
- **Transaction** = Atomic interaction.

Service descriptions

- Each service is identified by a
 - (1) triggering event and a
 - (2) delivered value (benefit).
- Each service may make assumptions about the environment.
- Do not give details about system behavior nor about communication channels with the system.
- Just describe the valuable effect that the system should have on the environment.
- A service may have a simple or a complex behavior; describe this later, using techniques from chapters 11 and 12.

Service descriptions are not system components

- For programmers, it is hard to see a piece of text *not* as a software component.
- A service description is not a software component;
- it is a description of something useful done by the software.

Main points

- Mission relates system functions to business goals.
- Function refinement tree relates services to mission.
- Services have a discrete beginning and deliver a value.
- Services may be non-atomic, consisting of many transactions.

Chapter 8. Entity-Relationship Diagrams

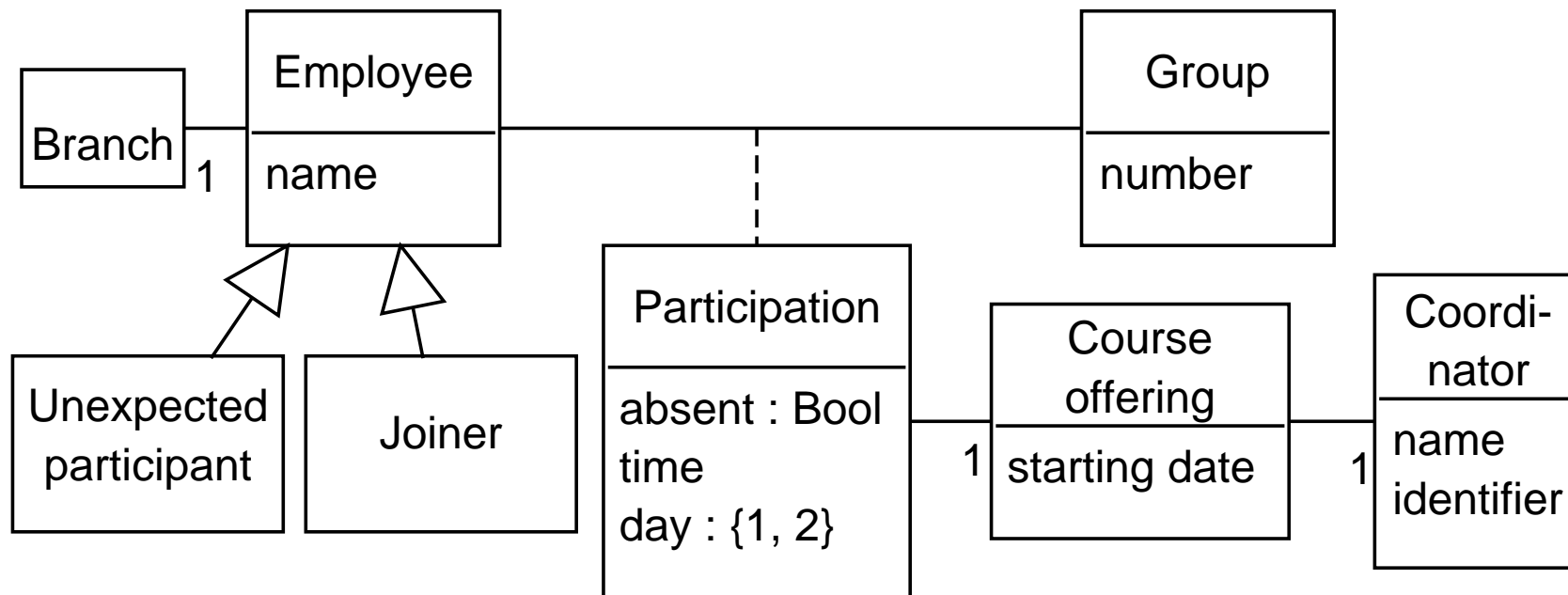
The **subject domain** of a software system is the part of the world talked *about* by the messages that cross the system interface.

To find the subject domain of a system

- Look at the messages received and sent by the system, and
- Ask what these messages are about.

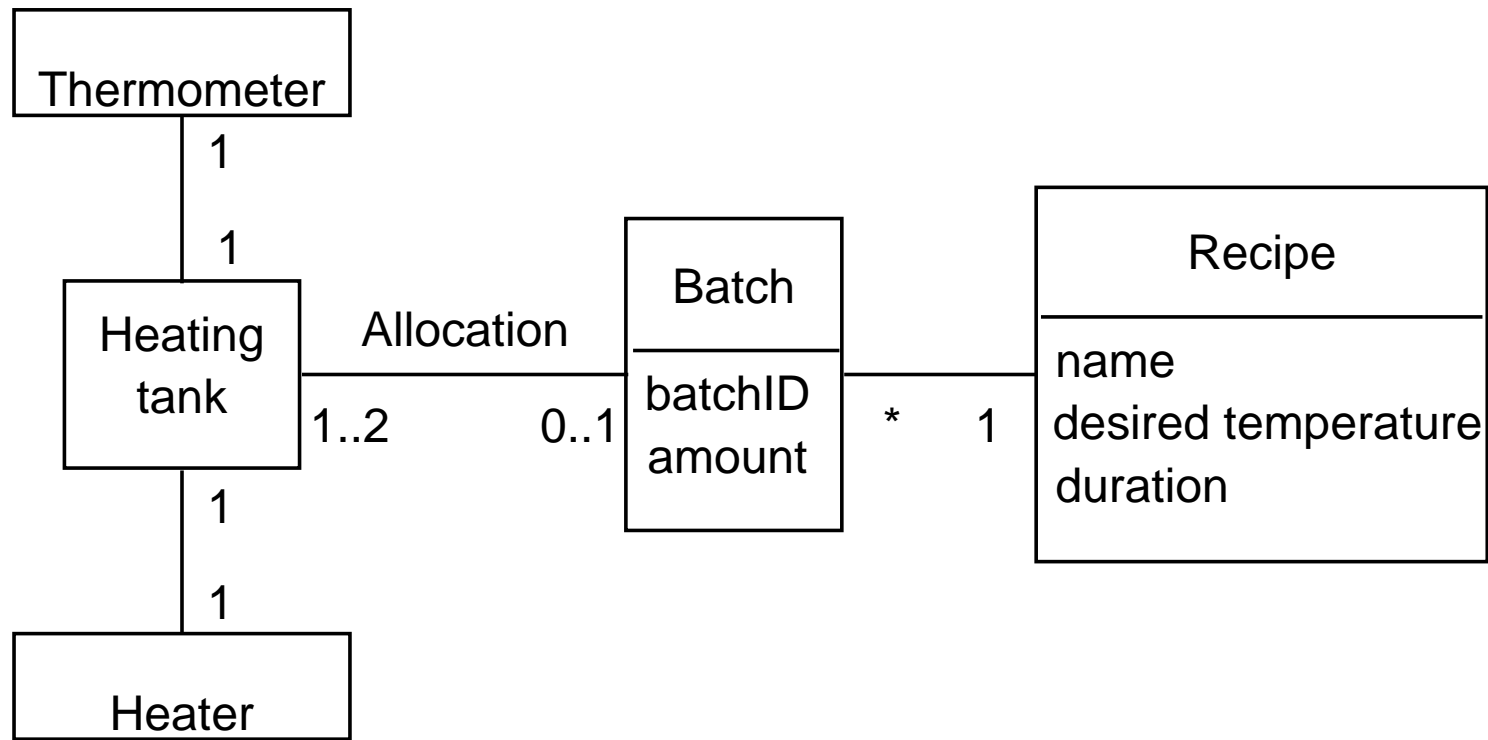
Document the meaning of these messages in a dictionary, supplemented by an ERD of the subject domain.

Example 1: Subject domain of TIS



These entities are the topic of interactions with TIS.

Example 2: Subject domain of heating controller

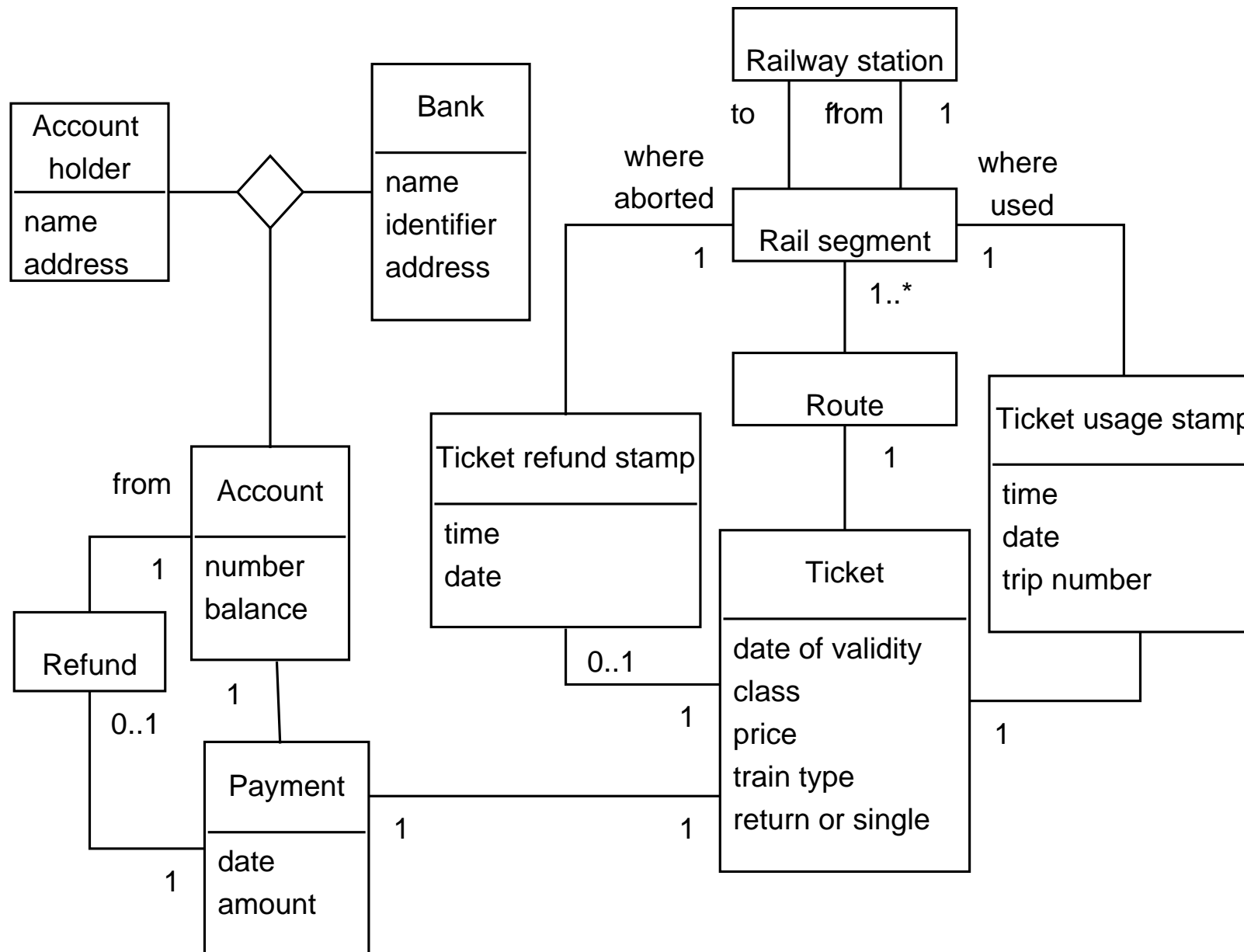


These entities are the topic of interactions with the controller.

Example 3: Subject domain of ETS

See next slide.

- These entities are the topic of interactions with the ETS.
- Some of them are lexical entities to be stored in ETS.



Syntax and semantics of ERDs

Bank

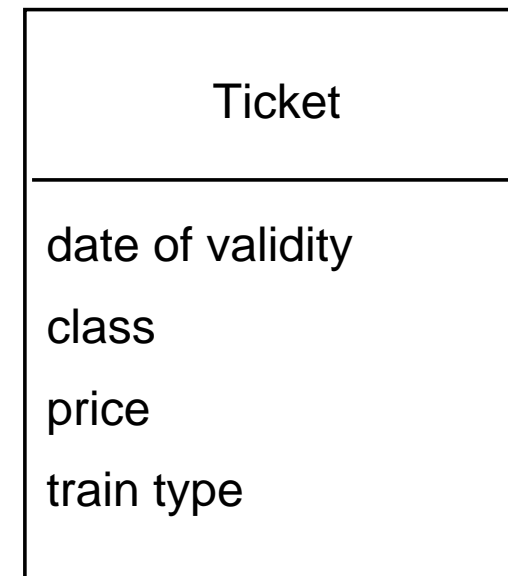
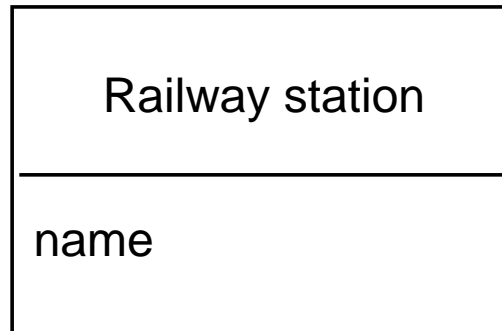
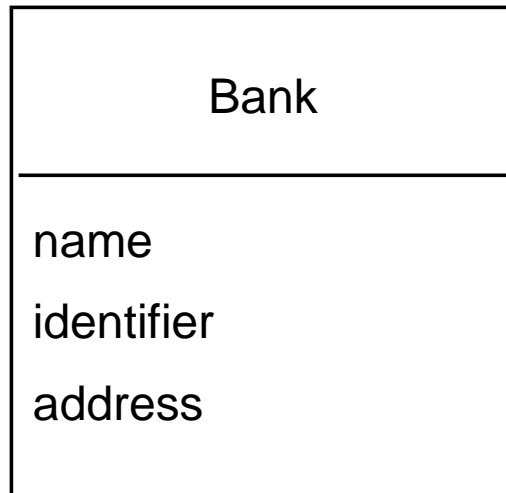
Railway station

Ticket

Entity type.

- Represented by a rectangle.
- **Extension** = All *possible* instances of entity type.
- **Extent** = All *existing* instances of entity type.
- **Intension** = All properties shared by all possible instances.
- **Defining intension** = Properties used to define the entity type. This is an abstraction (simplification) of the full, informal meaning.

Attributes



- Properties of entities.
- Can be listed in a compartment directly below the type name compartment.

Question: Which properties of a railway station did we describe in this diagram?

Relationships

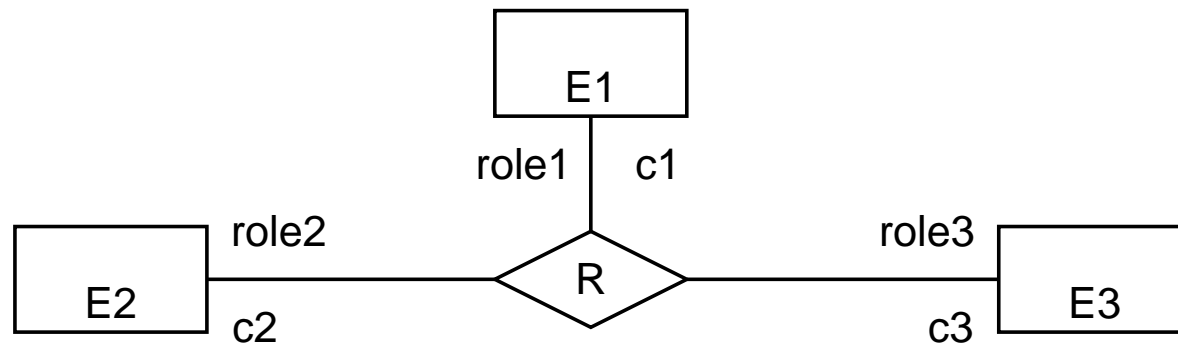
- A set of tuples of entities.
- Arity is the number of related entities. Binary, ternary, etc.
- Everything in the world is related to everything else!
- We only describe a relationship if we want to express relative cardinality properties: How many entities of one type can exist for each entity of another type.

Binary relationships



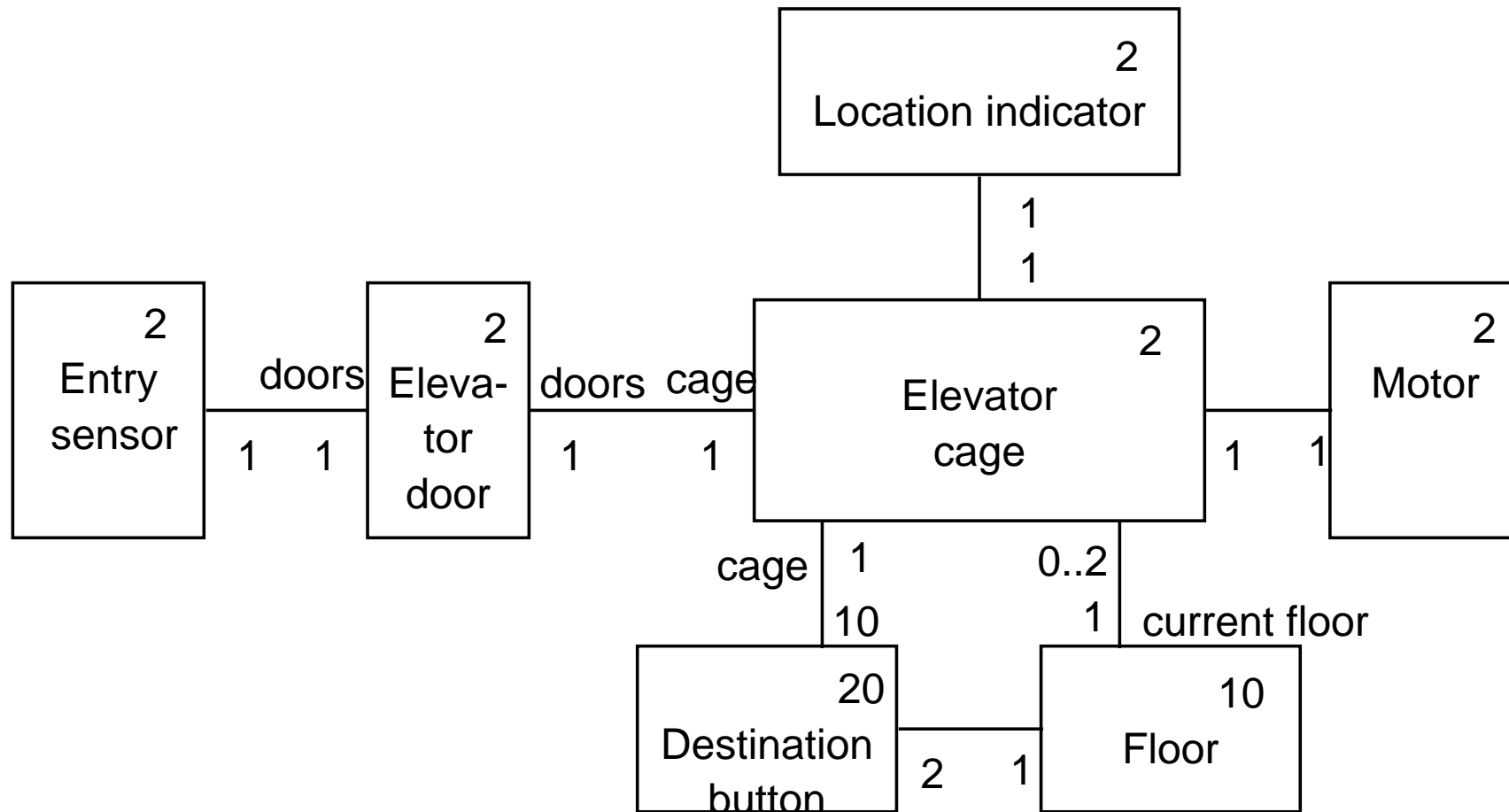
- Binary relationship represented by a line.
- Relationship name can have a read direction.
- Role names indicate the role an entity plays in a relationship.
- Relationship name and role names can all be omitted if this does not lead to ambiguity.

Relationships of higher arity



- Relationships with arity ≥ 3 represented by a diamond.
- Name must be independent from direction of reading.

Cardinality properties: Elevator example

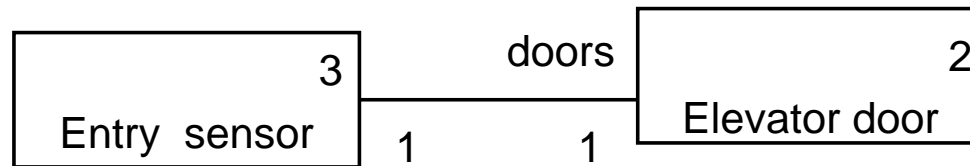


The meaning of cardinality properties

- A snapshot of the subject domain is the state of the subject domain at one point in time.
- Cardinality properties are snapshot properties.
- An absolute cardinality property says how many instances of an entity can exist in a snapshot.
- A relative cardinality property says how many entities can exist relative to some existing entity.

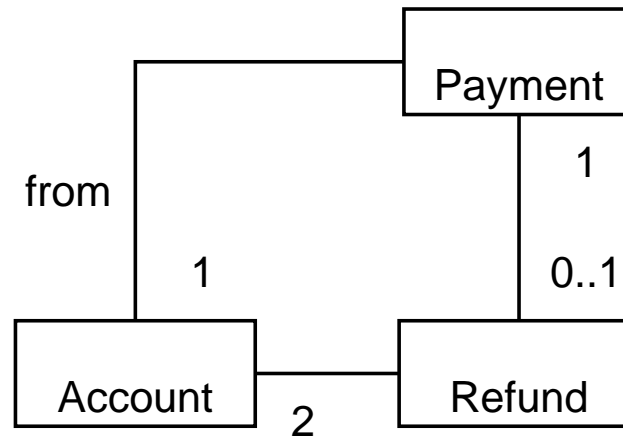
To find a cardinality property, imagine looking at an *arbitrary* state of the domain and ask how many instances of some entity type can exist in that state.

First example of inconsistent cardinality properties

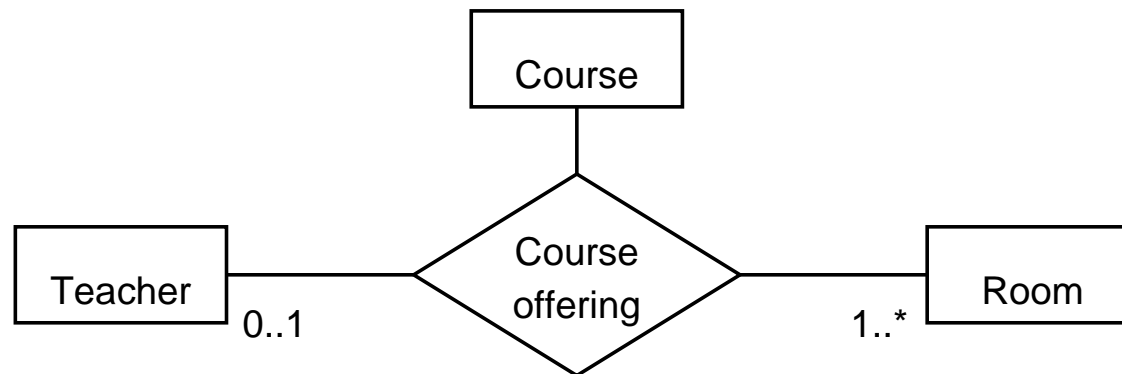


Second example of inconsistent cardinality properties

For p : Payment,
 $p.$ from = $p.$ refund. account



Ternary relationships

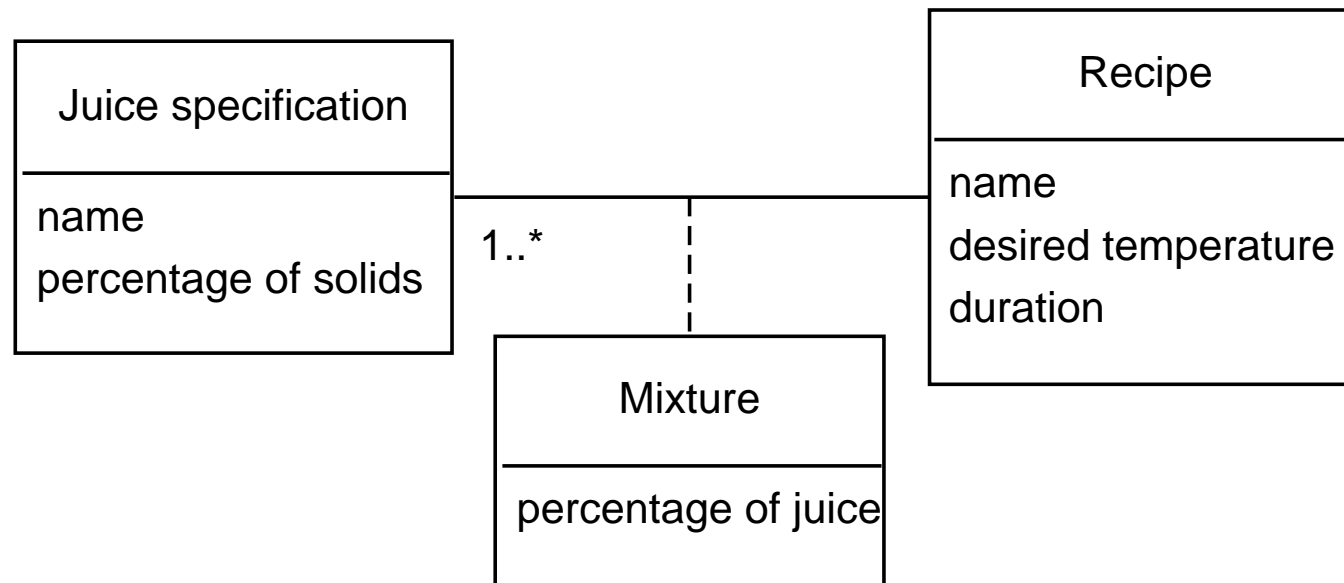


Cardinality properties of relationships with arity ≥ 3 are hard to understand.

Questions

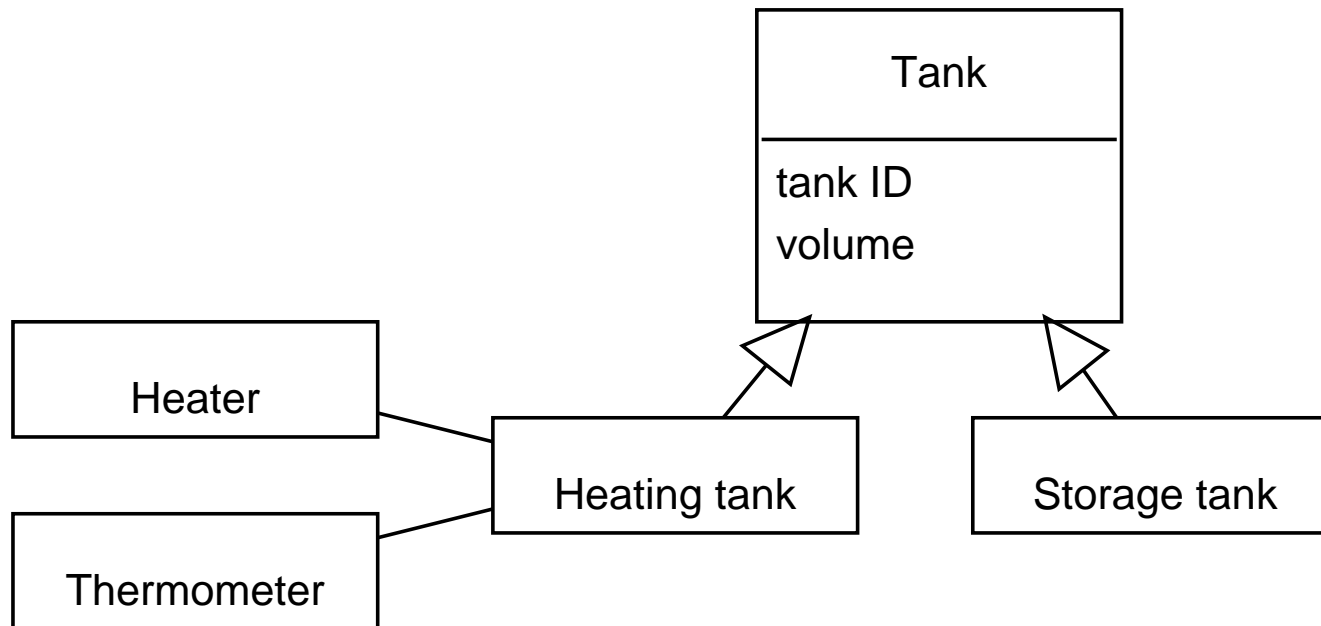
1. Can a course have more than one teacher?
2. Can a teacher give several courses?

Association entities



- An association entity is a relationship with attributes.
- Each instance is really a relationship. It is identified by a tuple of component identifiers.

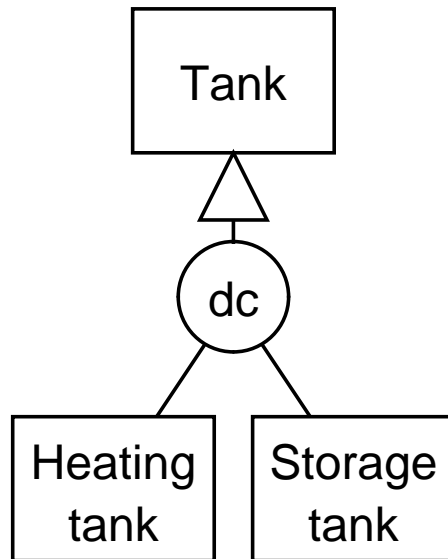
Generalization



Characteristic of specialization:

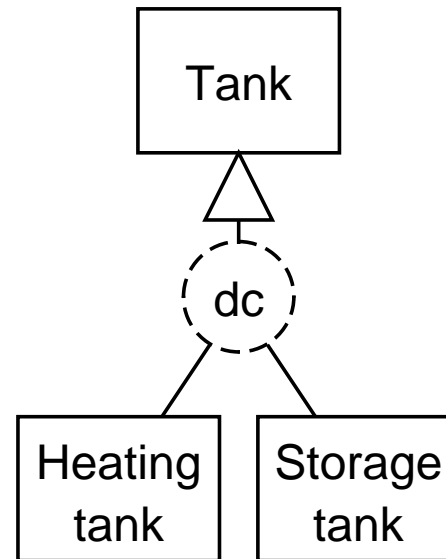
- Instance of a subtype is instance of a supertype.

Dynamic versus static specialization



Static specialization:

Subtype *extensions* are disjoint and cover supertype *extension*.



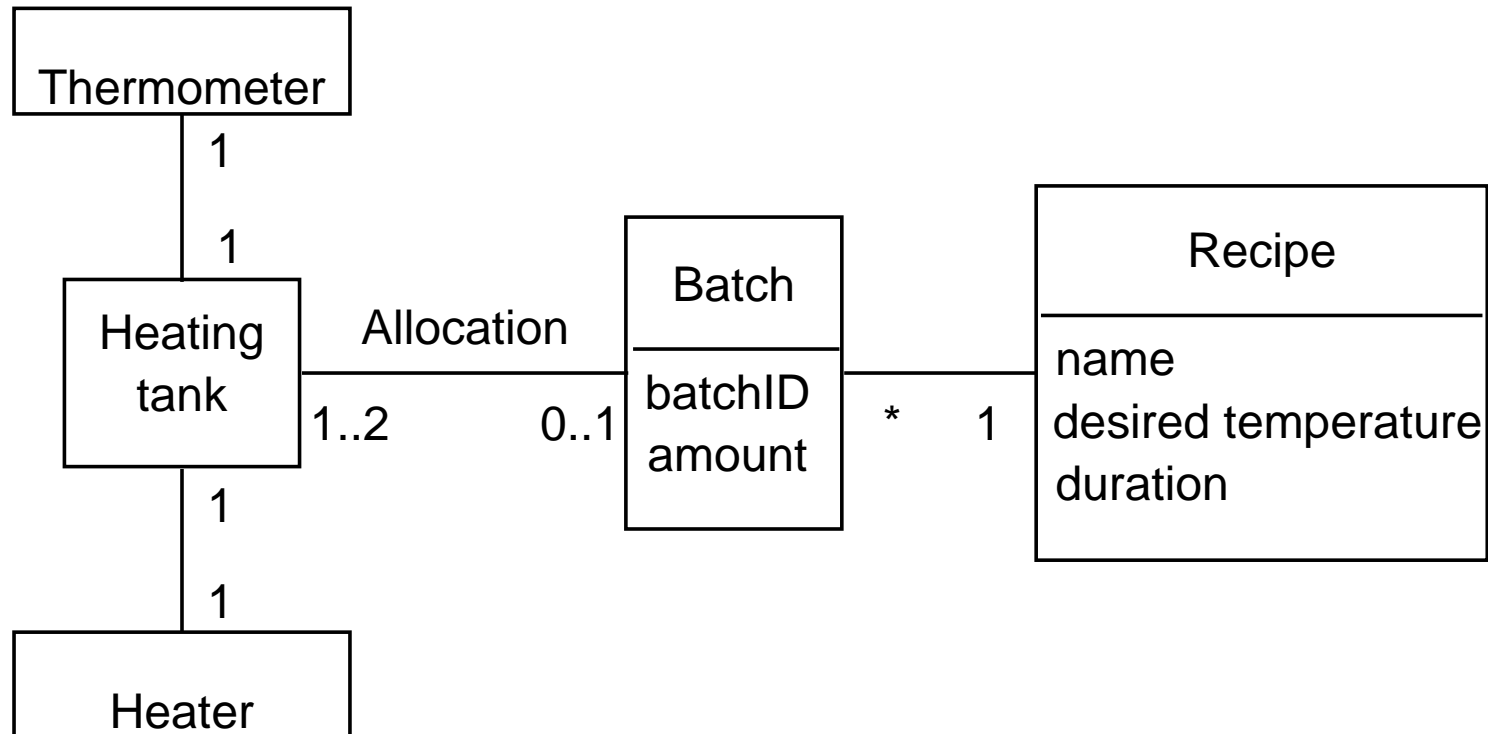
Dynamic specialization: Subtype *extents* are disjoint and cover supertype *extent*.

NB. If any heating tank can become a storage tank and vice versa, then all three extensions are equal! Why?

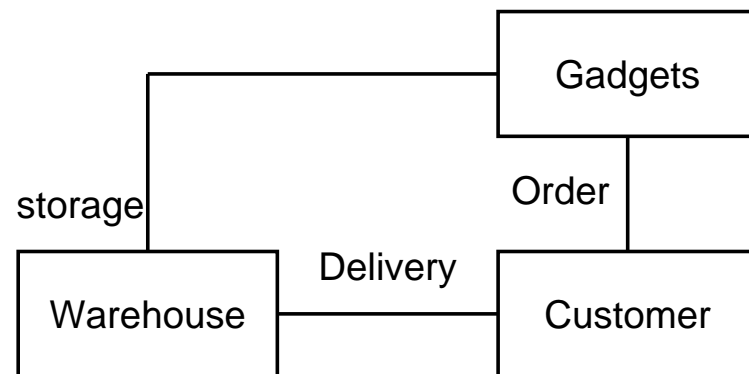
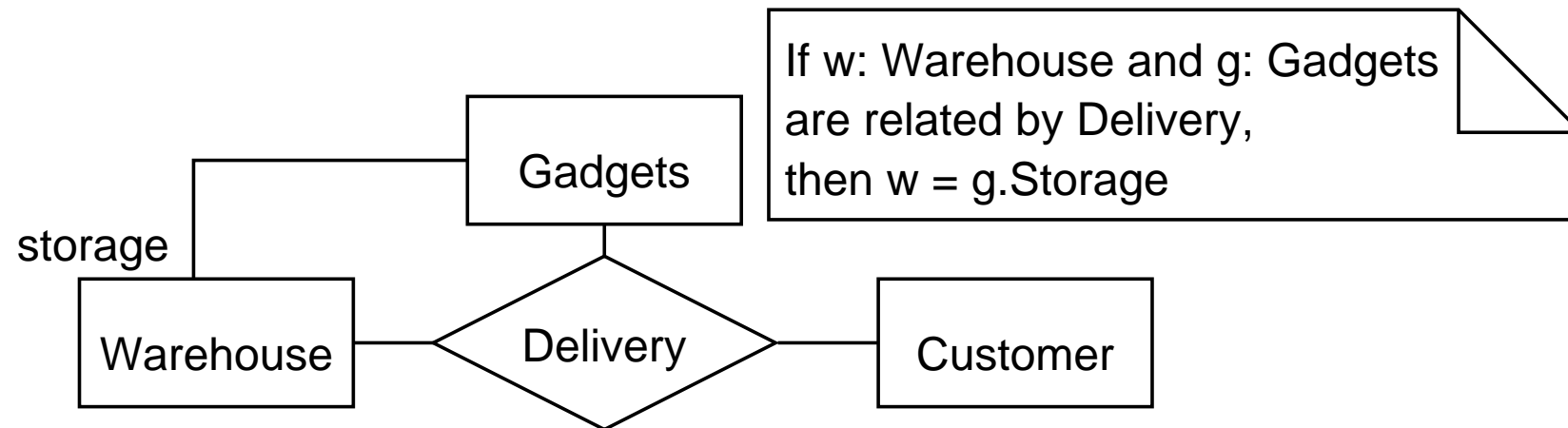
Main points

- We make an ERD of the subject domain, not of databases.
Even virtual entities are interesting only because they are part of the subject domain.
- ERD represents identification and classification of entities; and cardinality properties.
Counting and classification are closely related. Class (= type) provides identification criterion.
- Classification can be static or dynamic.

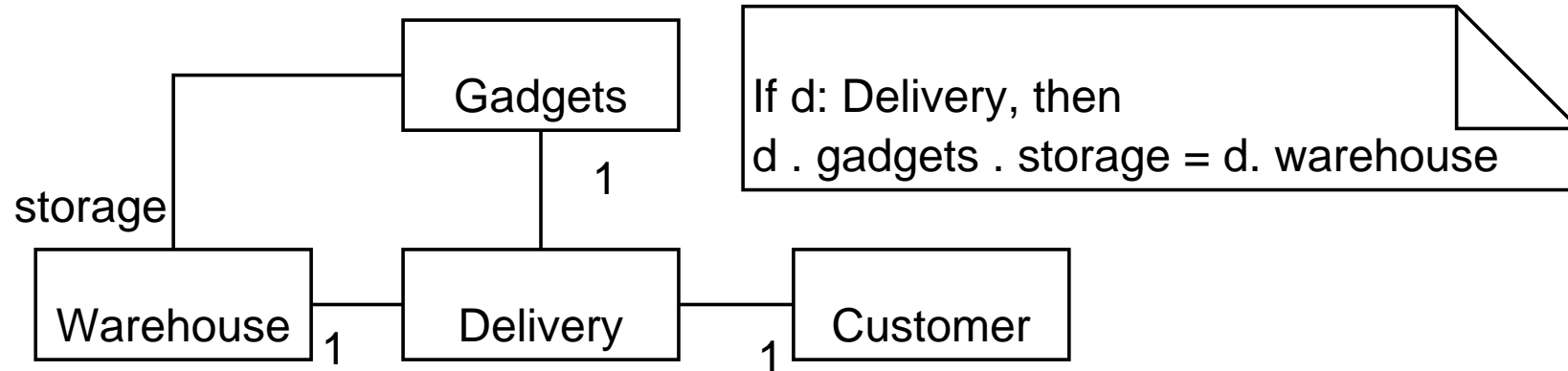
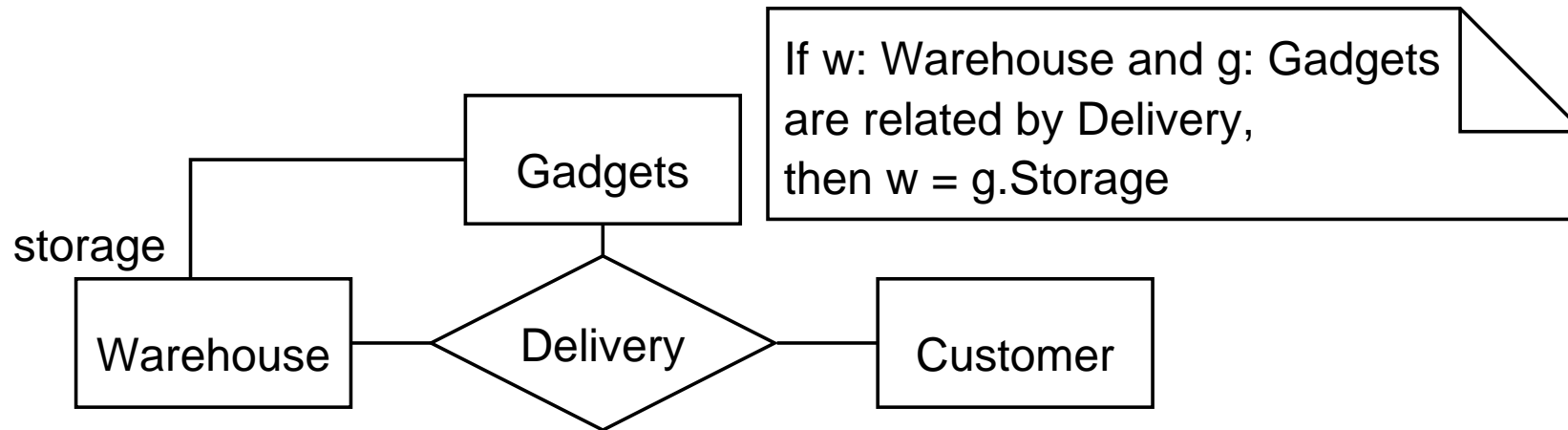
Questions



- Suppose we interpret cardinalities historically: “the number of instances related to in a lifetime”. Change the cardinalities to historical ones.



- Are these diagrams equivalent? Beware of the connection trap.



- Are these diagrams equivalent?

Chapter 9. ERD Modeling Guidelines

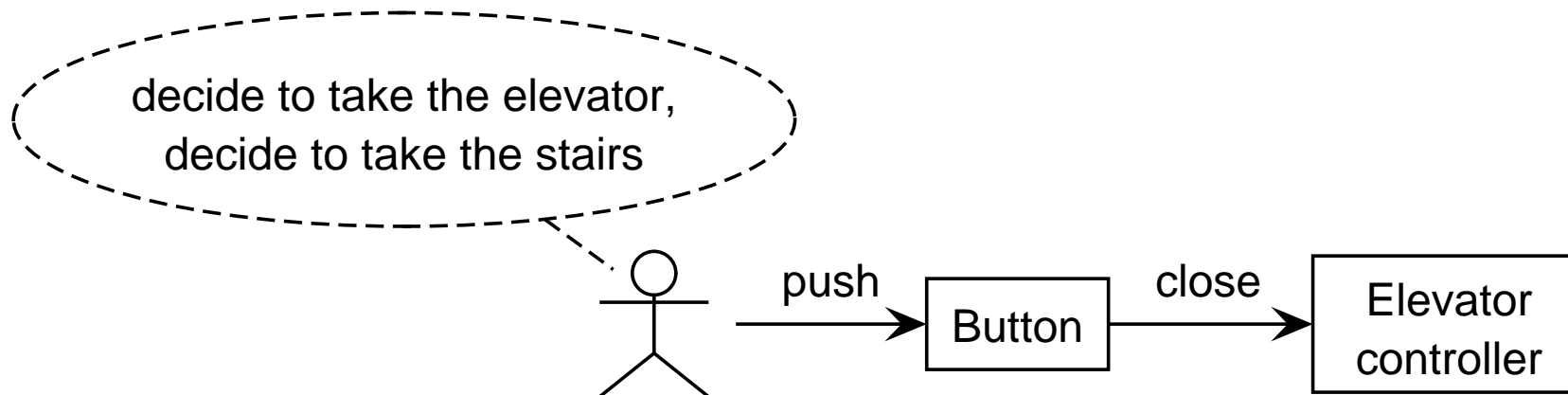
- ERDs can be used to declare any set of entity types and their cardinality properties.
- In this course we use ERDs only to represent the structure of the subject domain.
- Guidelines to determine the boundary of the subject domain are relevant for this kind of use of ERDs only.
- All the other guidelines are relevant for all possible uses of ERDs.

Subject domain boundary

The subject domain consists of entities and events. To find the boundary:

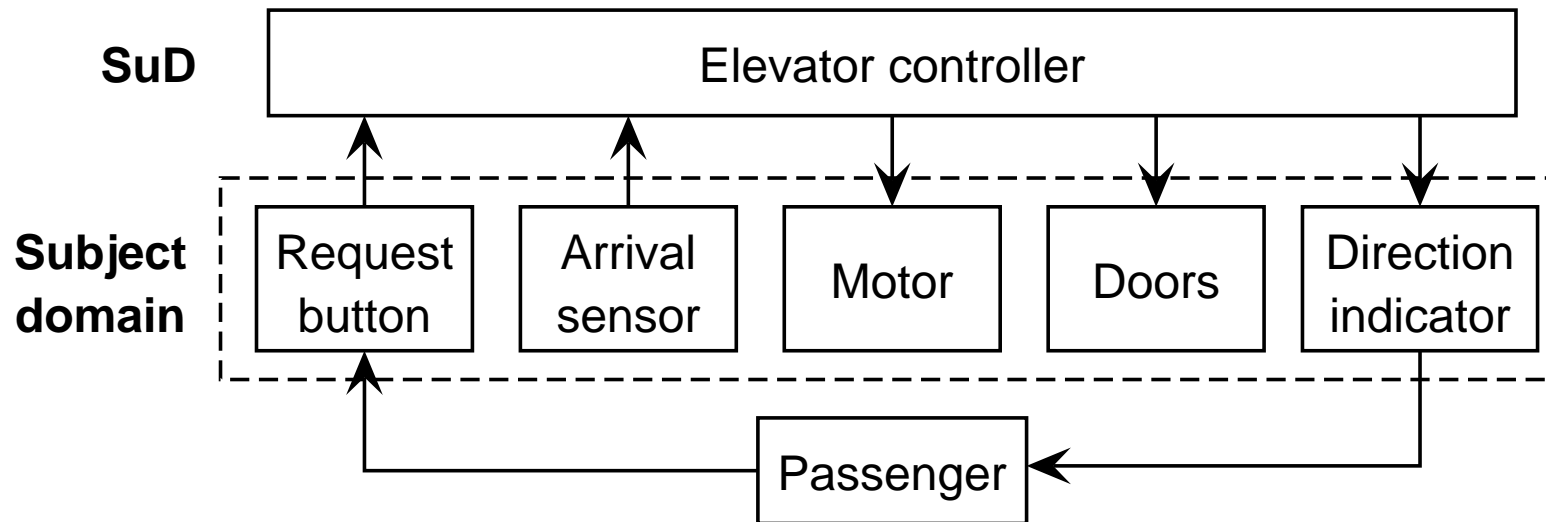
- ✓ Roughly: What entities and events do the service descriptions refer to?
- ✓ More precisely: What are the messages entering and leaving the system *about*?
- ✓ Look for
 - Physical bodies
 - Devices
 - (Parts of) organizations
 - Conceptual entities
 - Lexical itemsand events in their life.
- ✓ Entities and events should be identifiable by the SuD.

Unobservable events



Connection between events of interest (decisions by passenger) and stimuli of controller are too weak to include passenger in subject domain.

Including unobservable entities in a diagram of external communications



- The subject domain contains devices with whom and about whom the elevator controller exchanges messages.
- However, we can include unobservable entities in a diagram that shows external communication channels (the context diagram).

Questions

- Which entities are in the subject domain of a railway travel planner?
 - Passenger, Station, Railway track, Trip segment, Route.
- Is a Passenger part of the subject domain of the Electronic Ticket System? Why (not)?

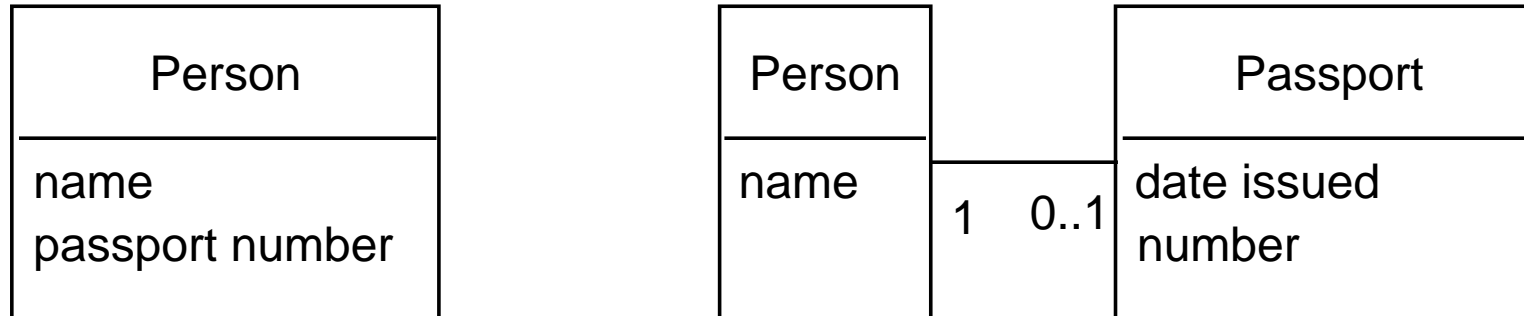
The answers are determined by

- (1) desired functionality of system and
- (2) choices in connection technology.

Entities versus attributes

- ✓ Entities are the things that the system talks about.
- ✓ Attributes are the things said about entities.
 - So if you want to store information about it, it is an entity.
- ✓ If information about it can change, it is an entity.

Example



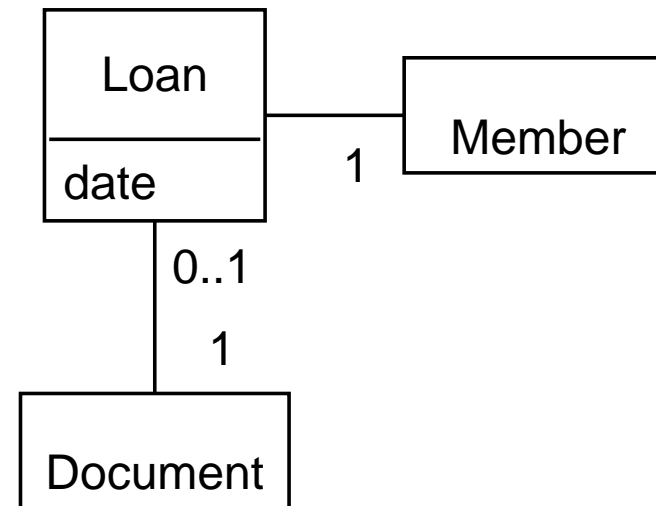
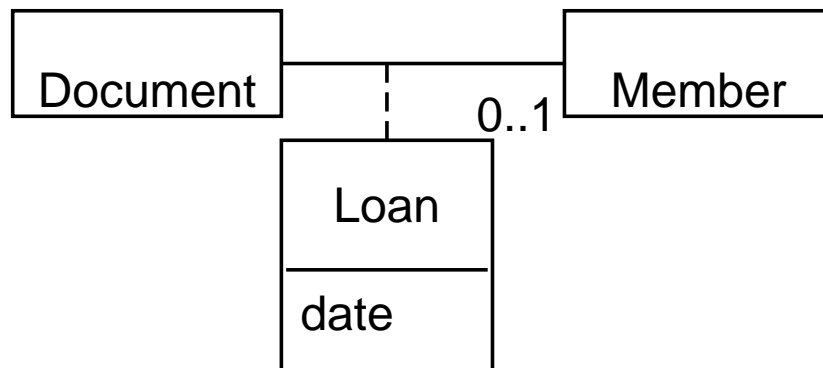
- Counting persons is not the same as counting passports.
- Persons are not created at the same time as passports.

So they are different entities. But which ERD is “correct”? That depends:

- If the SuD must be able to talk about persons without passports, or
- about persons with multiple passports (change cardinality property for that),
- then we must include a separate passport entity type.

Entities versus relationships

- ✓ Relationships are identified by their components.
So they are counted differently from the way entities are counted.

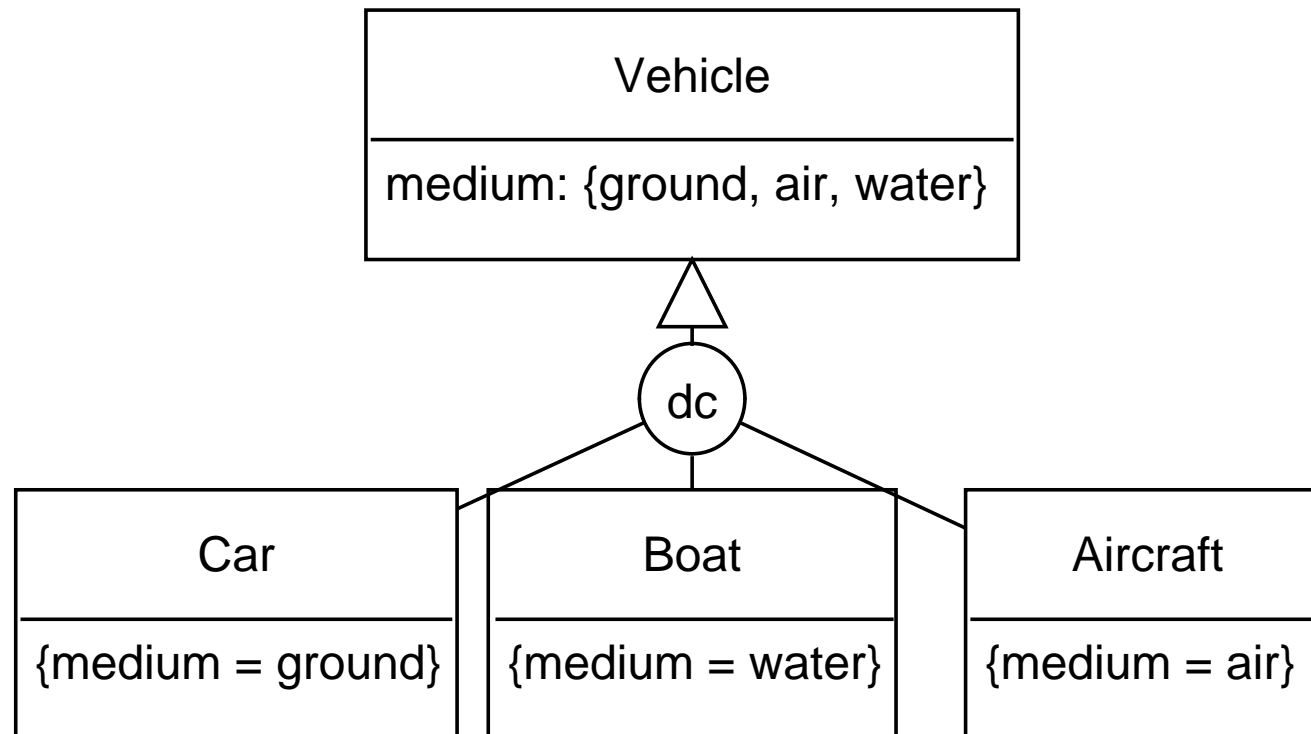


Several loans may co-exist!

Not what we intend

Taxonomic structures

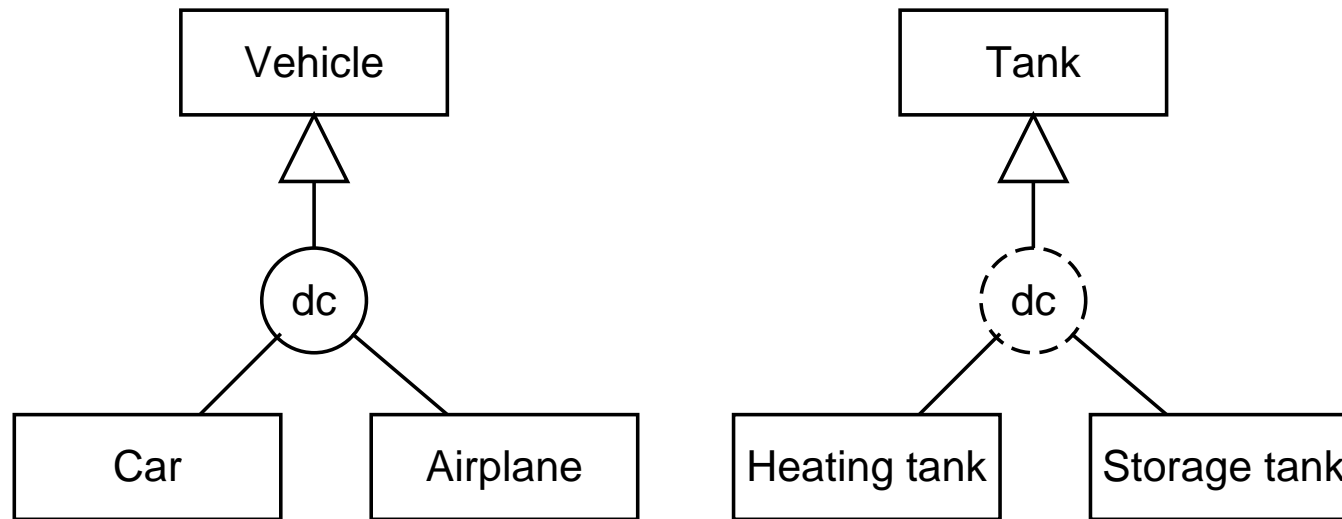
✓ Use a specialization attribute.



Any subtypes missing?

Do we need these subtypes?

Static versus dynamic specialization



According to this ERD, if you are born as a car, you are always a car. But heating tanks can become storage tanks and vice versa.

- ✓ Creating a static subtype instance is creating a supertype instance.
- ✓ Creating a dynamic subtype instance may not involve creating a supertype instance.

Classification and identification

- ✓ A type definition should provide a recognition criterion and an identification criterion.
 - The recognition criterion for `Car` gives us the answer to this question “Is this a car?”
 - The identification criterion gives us the answer to the question “How many cars do we have here?”

To find the identification criterion of type `C`, it is often useful to look at the creation event of an instance of `C`.

Questions

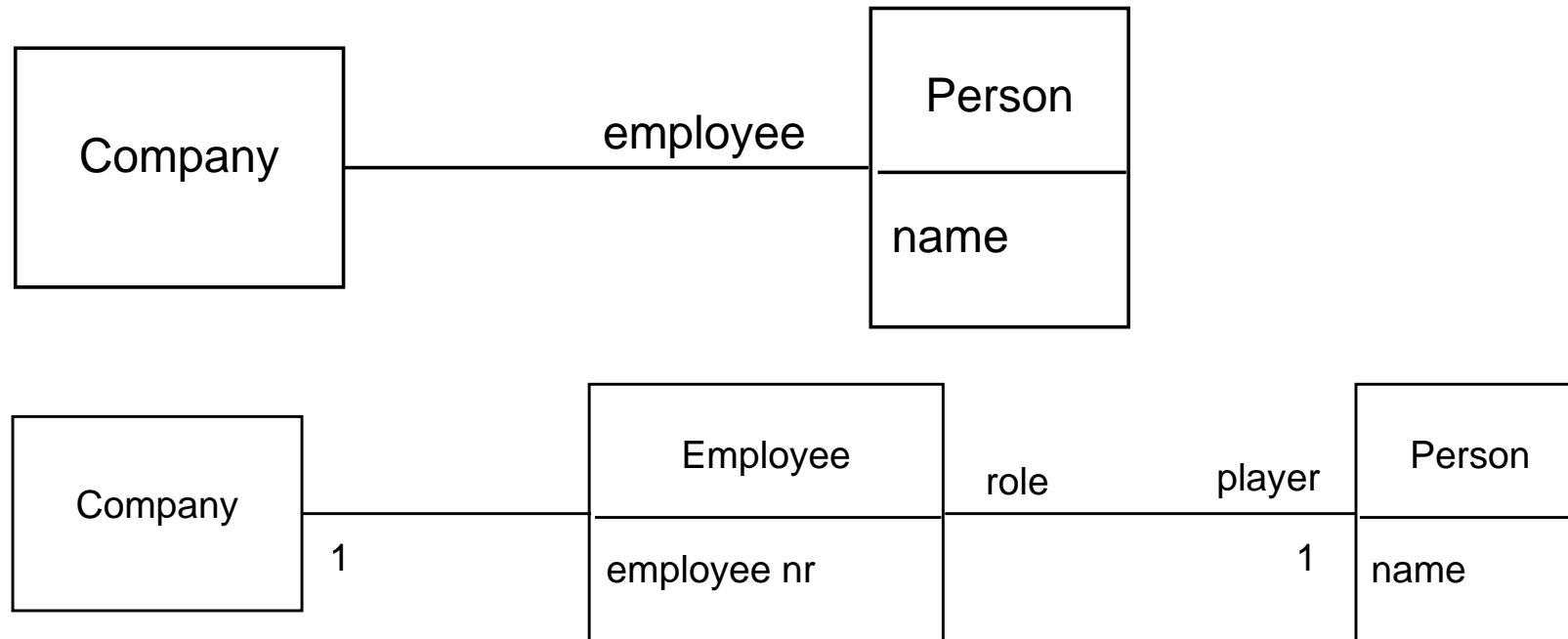
Give recognition and identification criteria for the following concepts:

- Person
- Company
- Passenger
- Elevator button
- Your PC
- The chair you are sitting on.

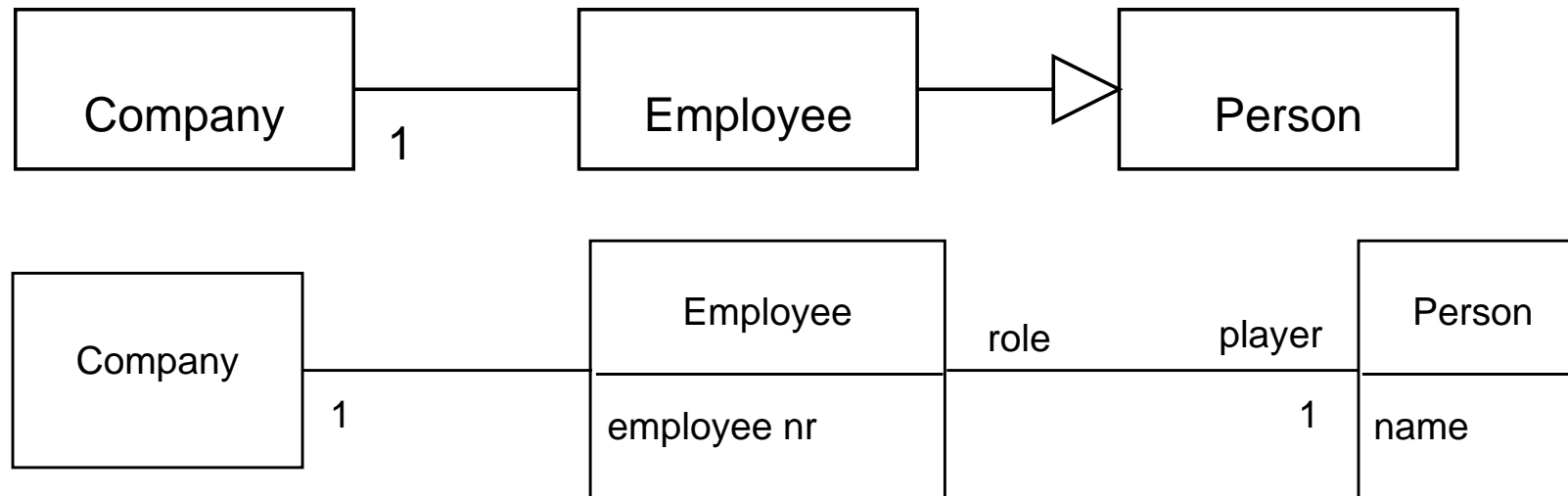
If an instance of C is in the subject domain, then the SuD is able to talk about a C , and so the SuD should be able to recognize and identify C 's!

Subtypes versus roles

If an entity can play several roles of the same type, we can turn the role into an entity type.

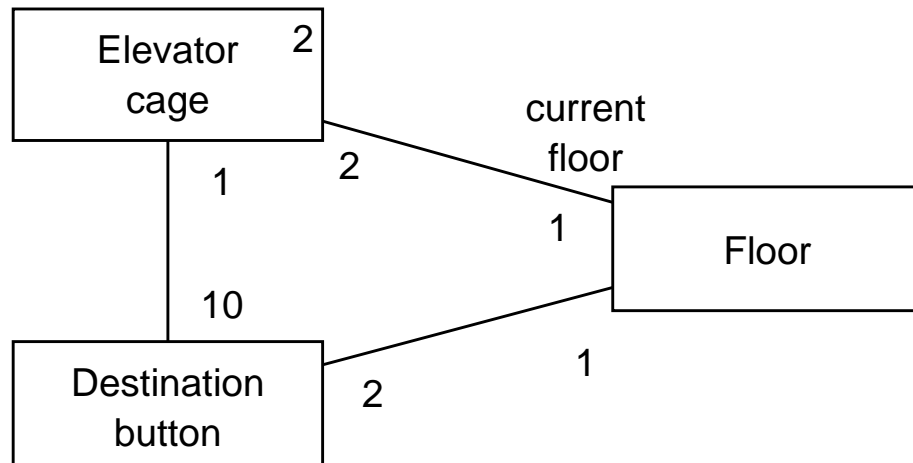


What is the difference between these two models?



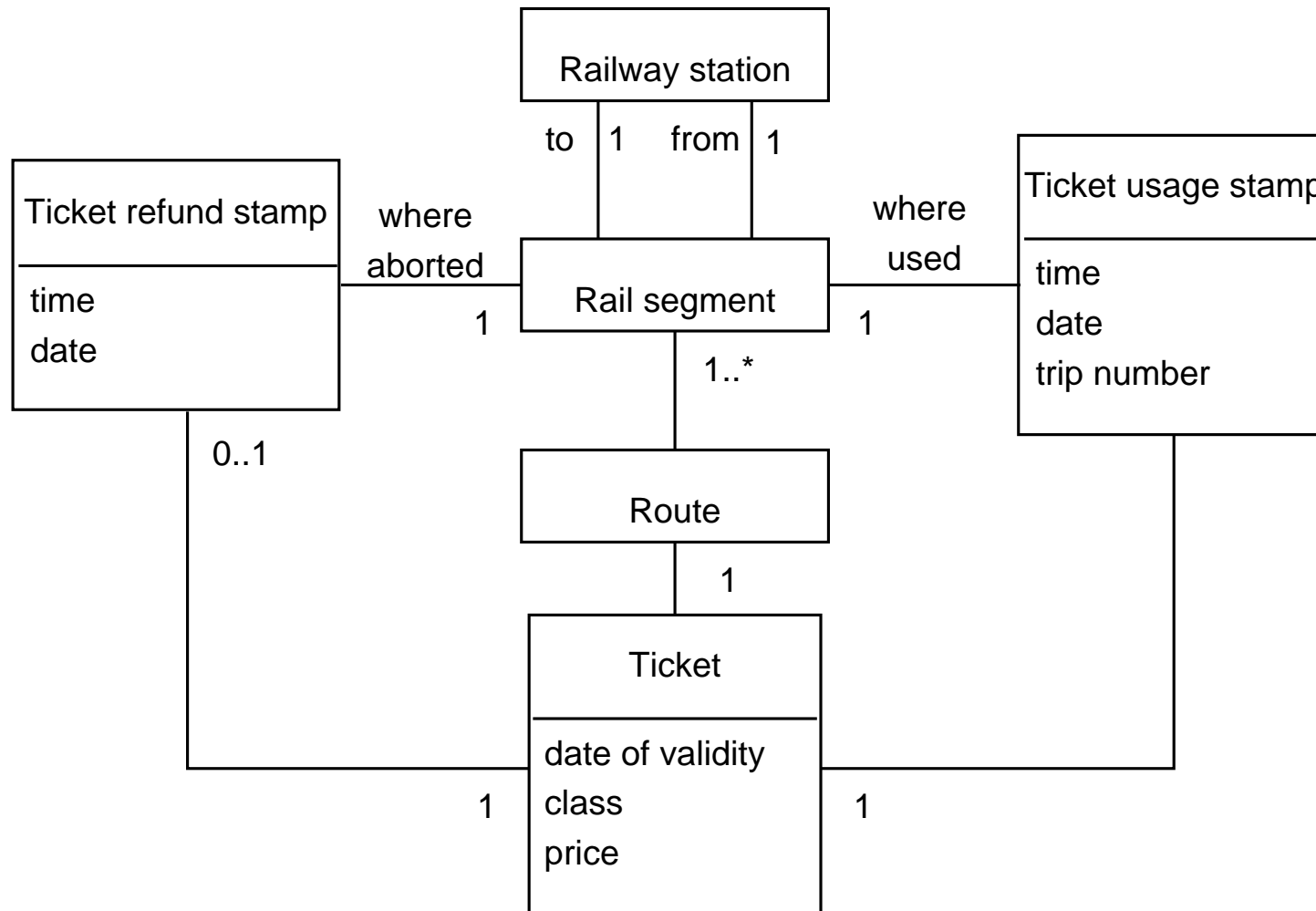
Validation of an ERD (1)

✓ Check consistency of cardinality properties.



If a diagram is not consistent, it cannot represent any subject domain.

Validation of an ERD (2)



Are all loops commutative (any starting / destination node)?

Validation of an ERD (3)

- ✓ Describe by elementary sentences and show to domain specialist.

At each moment:

- A batch of juice has exactly one recipe.
- A recipe may be applicable to any number of batches.
- At any point in time, a heating tank has at most one batch allocated to it.
- At any point in time, a batch is allocated to either 1 or 2 heating tanks.
- Each heating tank has exactly one heater.
- etcetera.

Validation of an ERD (4)

✓ Describe snapshot and show to domain specialist.

Ticket	Route	Segment	Usage stamp	Refund stamp
(t1, April 2)	Return Enschede- Apeldoorn	Enschede- Apeldoorn	March 31	None
(t1, April 2)	Return Enschede- Apeldoorn	Apeldoorn- Enschede	April 1	April 1

Flaws in the model:

- Ticket cannot be used before bought!
- No usage and refund for the same segment!
- Return must be on the same day!

These are subject domain properties.

Validation of an ERD (5)

✓ Check against messages entering and leaving the system.

Or against service descriptions:

- | |
|--|
| <ul style="list-style-type: none">• Name: Sell a ticket.• Triggering event: Traveler requests to buy a ticket.• Delivered service: Allow a traveler to buy a ticket at any time and place chosen by the traveler. |
| <ul style="list-style-type: none">• Name: Show a ticket.• Triggering event: Traveler requests to view a ticket.• Delivered service: Display ticket attributes to the user. |

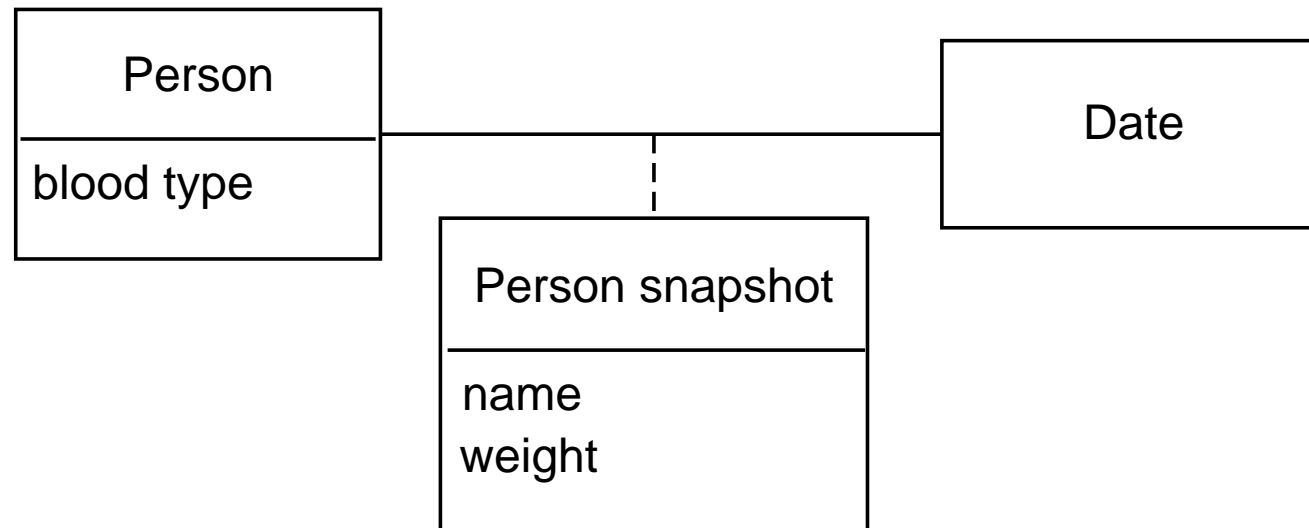
Why is “Traveler” not an entity in the subject domain?

Main points

- Subject domain bounded by the topic of the conversation with SuD.
- Entities have contingent properties, attributes do not.
- Relationships are identified by their components.
- Use specialization attribute.
- Static and dynamic specializations are distinguished by what happens at creation time.
- Classification and identification closely connected.
- Roles can be reified to entity types.
- Validate your ERD: consistency, elementary sentences, snapshots, check against interface of SuD.

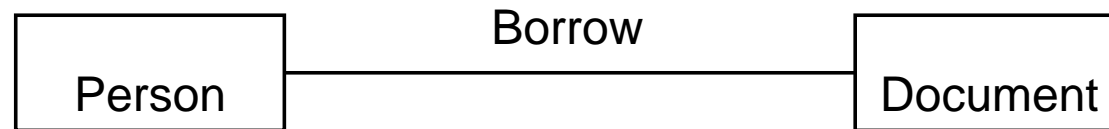
Chapter appendix (slides only): describing
histories and events in an ERD

Histories

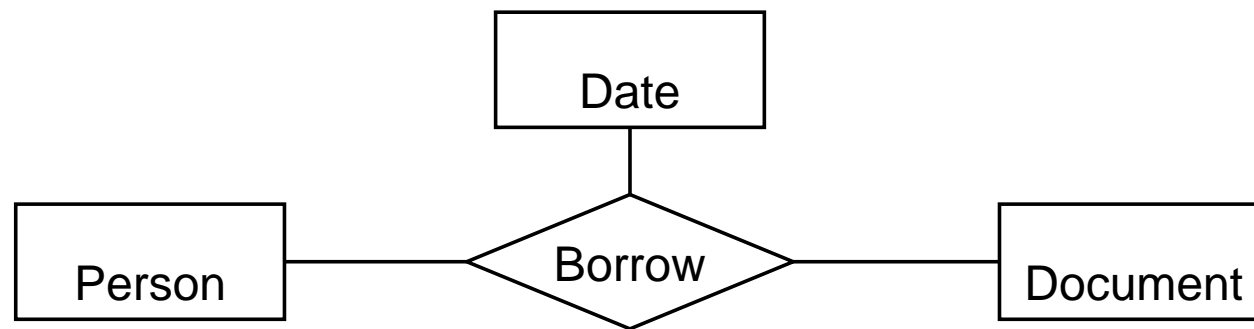


- Fixed attributes in the **Person** type box.
- Date-dependent attributes in an associative entity which represents the person at a certain date.

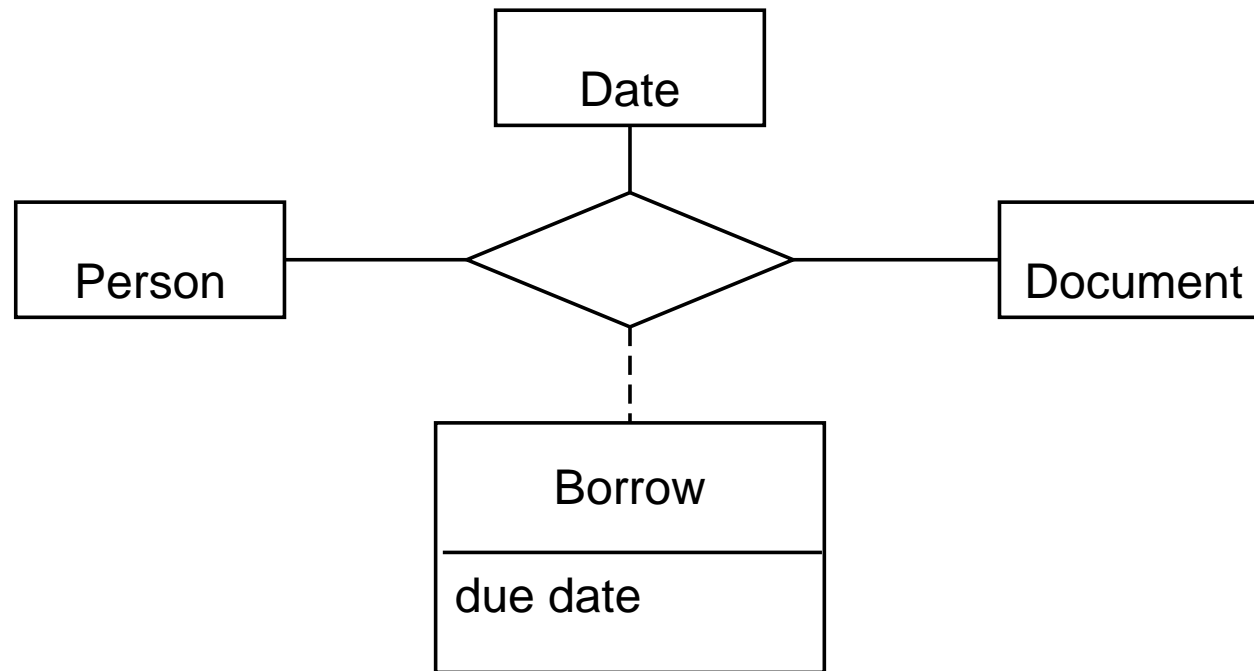
Events



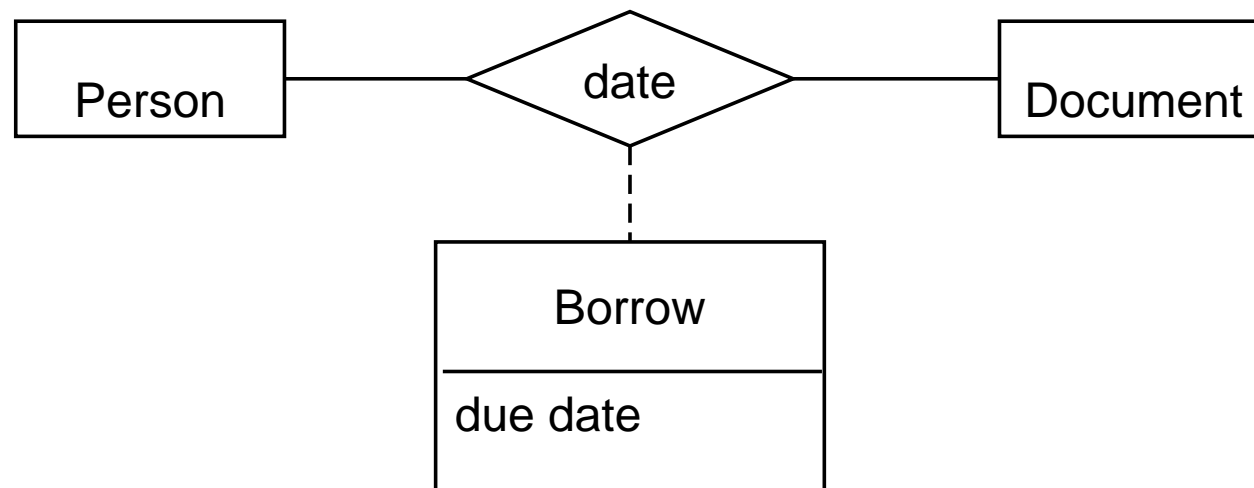
- Borrow is an event.
- An instance of Borrow exists for one moment only.



- Borrow is a dated event.
- An instance of Borrow represents the fact that at a certain date, a person borrowed a certain document.
- An instance of Borrow has identifier (p, dt, dc).
- We can include a time indication in the date too.



- Now Borrow has an attribute.



- Convention that abbreviates the diagram on the previous slide.

Chapter 10. The Dictionary

The dictionary documents the meaning of important terms:

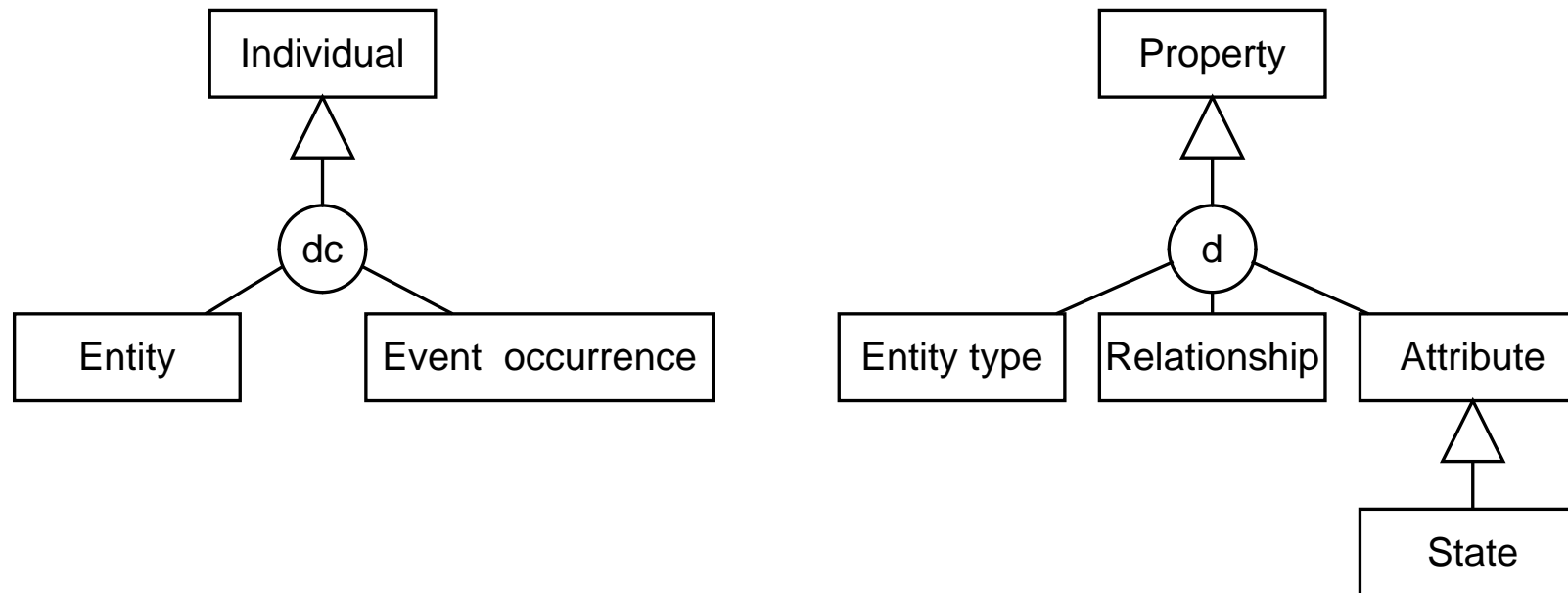
- Words used in service descriptions
- Words used in messages that cross the external SuD interface
- Domain-specific jargon
- etc.

The subject domain ERD is a visual supplement to the dictionary, that adds some precision to terms that refer to subject domain entities.

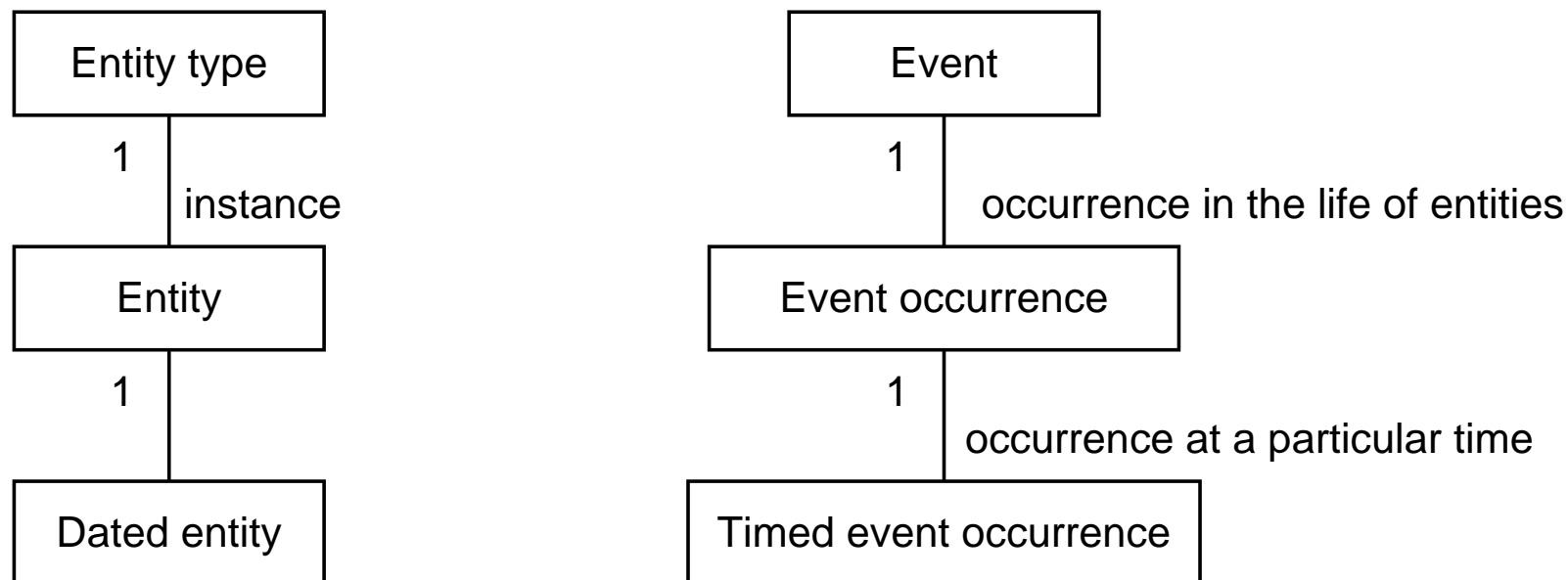
In our dictionary, we use only a few syntactic categories of terms, that are motivated by our domain ontology.

Domain ontology

- **Domain** is part of the world treated as a whole.
- **Ontology** is a metaclassification.



Individuals are entities and event occurrences



- No reincarnation of entities.
- Event occurrence can occur at many times.
sell(coffee, 0.2 l) is an occurrence of event **sell** and it can occur at times 9:00, 9:02, etc.

Syntactic categories

- **Identifier.** Unique proper name.
- **Predicate name.**
 - *Entity type name.*
 - *Relationship name.*
 - *State predicate name.* Boolean property.
- **Attribute name.**
- **Event name.**

Definitions from the ETS specification

- **Railway station.** Entity type. Entity used to bound *rail segments*. May consist of platforms where passengers can enter or leave a train. Can also be a mathematical point used to bound a segment. Examples: The point where the line of rail passes the Dutch-German border; one can buy a ticket to that point. Can also be a collection of physical stations. Example, a ticket to “Berlin U-Bahn” is a ticket to any subway station in Berlin.
- **Rail segment.** Entity type. A shortest path between two Railway stations. Segments are directed and are identified by the two stations they connect. So all shortest paths through the rail network from A to B are considered the same segment, called AB; and all shortest paths from B to A are the same segment, called BA; and AB and BA are two different segments.
- **Route.** Entity type. A path through the rail network that consists of a connected series of *rail segments*, where each segment occurs at most once in the route.

Definitions from an elevator control specification

- **planned direction(c: Elevator_cage)**. Attribute. The preferred direction in which c will depart after closing its doors. If there are requests to be served higher and lower than the current floor of the elevator cage, then it will depart in the planned direction.
- **Round trip time**. The time in seconds for a single car trip around a building from the time the cage doors open at the main terminal until the doors reopen when the cage has returned to the main terminal floor after its trip [Barney & dos Santos 1977].
- **Atfloor(b: Request button, c: Elevator cage)**. Predicate. c
▪ $\text{current_floor} = \text{b.floor}$.
- **continue(c: Elevator cage)**. Action. Term is applicable only if c
▪ $\text{planned_direction} \neq \text{none}$.
 - If $\text{c.planned_direction} = \text{up}$ then $\text{start_up}(\text{c.motor})$.
 - If $\text{c.planned_direction} = \text{down}$ then $\text{start_down}(\text{c.motor})$.

The motor of c is started in the planned direction of c.

Path expressions

Definitions may use path expressions, that refer to paths through the ERD.

See section 10.3 for syntax of path expressions.

Extensional and intensional definitions

- **Extensional definitions** list a few instances of the concept.

Easy to give. But do not define an intension.

- **Intensional definition** lists the defining properties shared by all instance of the concept.

Difficult to find.

Open-textured terms have no intensional definition.

- ✓ Define these by giving a few examples, a sketch of the intent, and indicating the procedure —if any— that determines whether an instances falls under the concept.

Examples: Boat, title, document, vehicle, house, elevator, ...

When to define a term

- ✓ To clarify a term.
- ✓ To indicate that the term is open-textured.
- ✓ To indicate that we attach little meaning to it.
- ✓ The term is absolutely obvious to all stakeholders —but they attach different meanings to it.

When not to define a term

- ✓ To raise a cloud of obscurity. (Bad idea, but frequent practice.)
- ✓ Definitions can also be found in technical documentation of devices. Don't repeat these.
- ✓ The term is absolutely obvious to all stakeholders —and they understand the same by it.

Definition by genus and difference

Without genus	With genus
A compiler translates source code into object code.	A compiler is a program that translates source code into object code.
A catamaran has two hulls.	A catamaran is a boat with two hulls.
A heating tank heats juice.	A heating tank is a tank with a heater and thermometer in which juice can be heated.
Joiners recently joined the company.	A joiner is an employee that recently has joined the company.
A ticket represents the passenger's right to make a trip by train.	A ticket is a lexical item that represents the passenger's right to make a trip by train.

- ✓ The genus provides the identification criterion of the entity type.
- ✓ The difference provides the recognition criterion of the entity type.

Operational definitions

An operational definition of a term gives a procedure, that can be followed by anyone, to determine whether or not the term is correctly applied to a given case.

- **Heating tank.** Entity type. A tank with a heater and thermometer attached. The heater can be recognized by red, blue and black wires leading up to it, and the thermometer by its rectangular shape.

This is a operational definition by genus and difference.

- **Round trip time.** The time in seconds for a single car trip around a building from the time the cage doors open at the main terminal until the doors reopen when the cage has returned to the main terminal floor after its trip.

Operational, but not as definition by genus and difference.

Abbreviations versus correspondence rules

- **Abbreviation** reduces the meaning of a word to the meaning of other words defined in the same dictionary.

To determine whether an individual is an instance of the defined concept, you do not need new observations but simply look up words in the dictionary.

- **Correspondence rules** relate a word to reality. The words in the definition are not defined in the same dictionary.

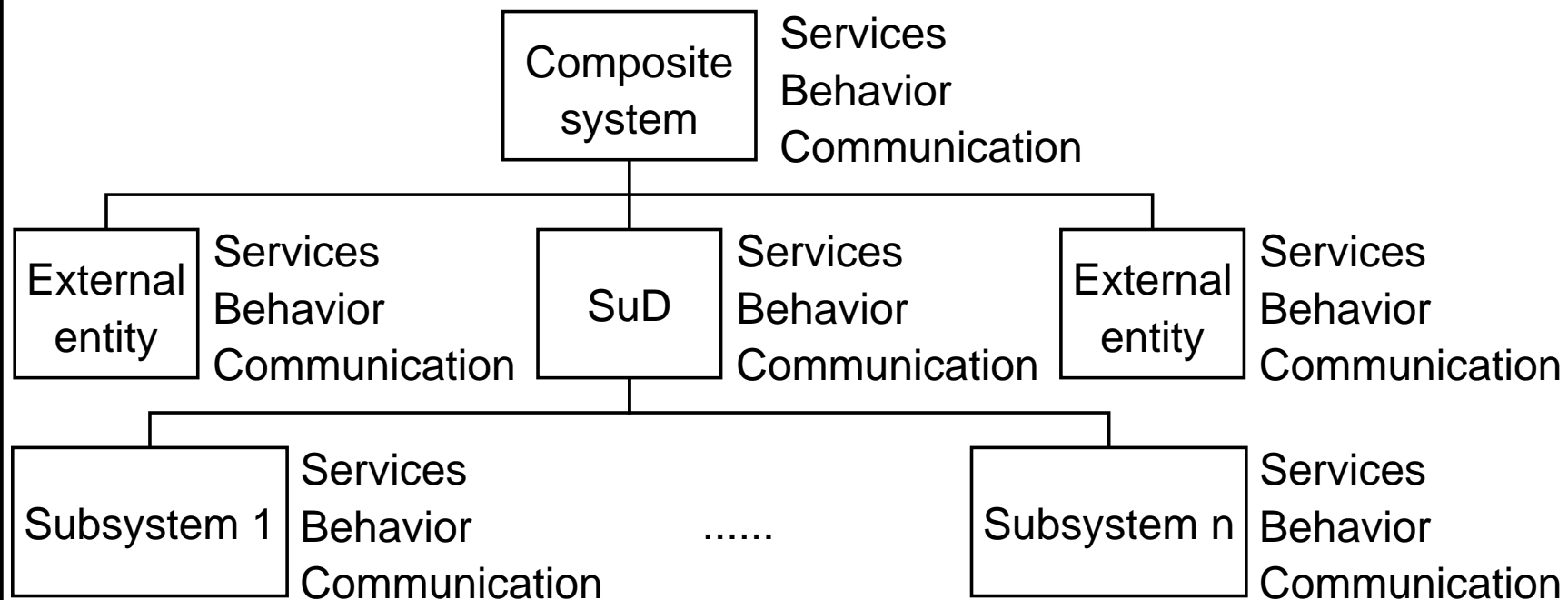
To determine whether an individual is an instance of the defined concept, you must make observations.

Which definitions in the examples given are abbreviations, and which are correspondence rules?

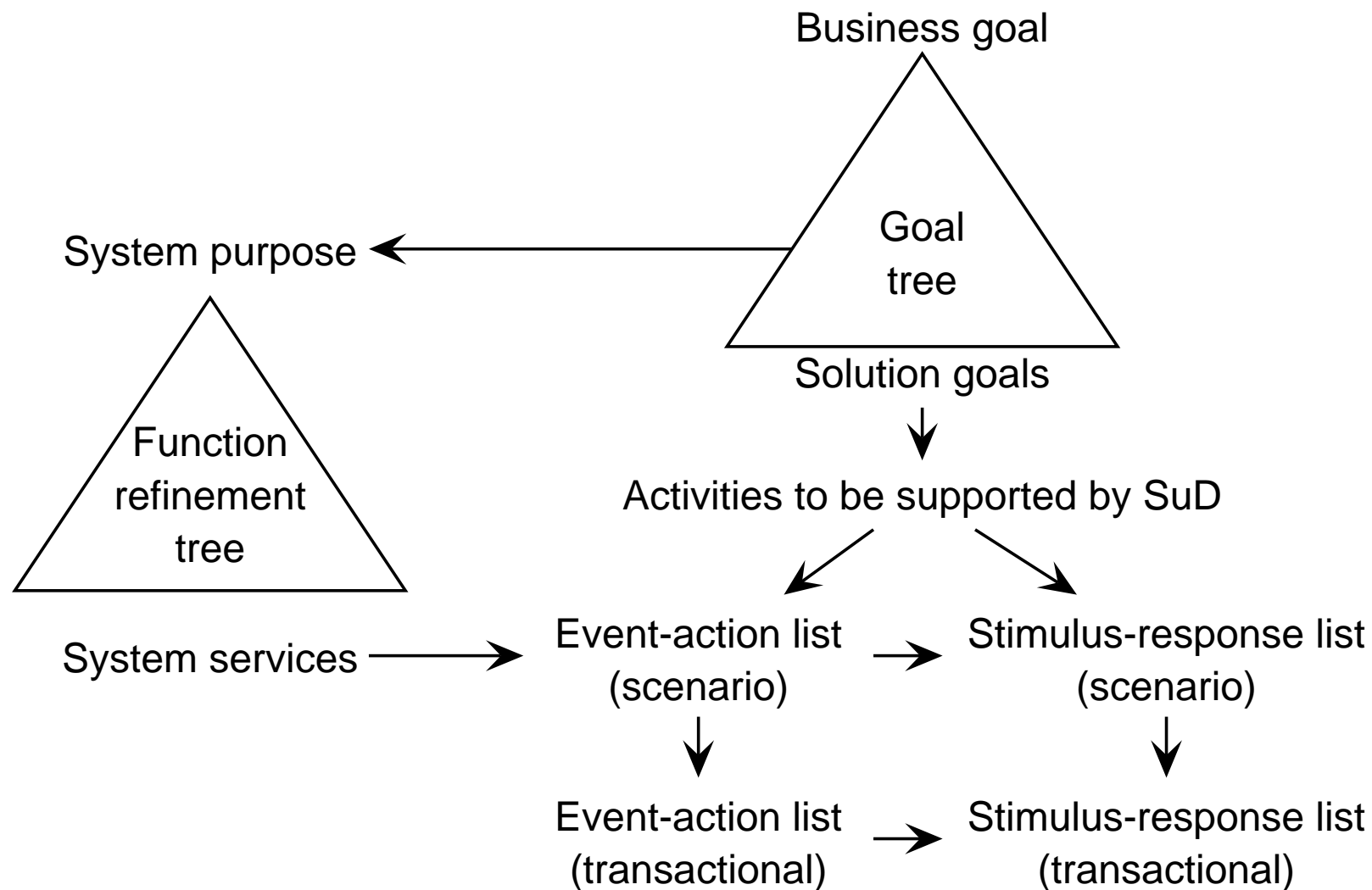
Main points

- The dictionary contains definitions of jargon, subject domain terms, and other important terms needed to understand the design specification.
- We can structure definitions as a taxonomic hierarchy: definition by genus and difference.
- Try to give operational definitions.
- Distinguish abbreviations (no new phenomenon described) from correspondence rules (link words to reality).

Part IV. Behavior Notations



Road map of ways to find behavior descriptions of the SuD



Example 1: Training Information System

We take the route through service descriptions to desired environment behavior.

Download Joiners service description

- **Triggering event:** Coordinator requests to download list of joiners from the personnel information system.
 - **Delivered service:** Download the list of people from the Personnel Information System who have joined the company since the previous training.
 - **Assumptions:** The data in the Personnel Information System reflects the situation accurately with a time lag of not more than one working day.
- Declarative service description! Not a scenario.

Example 1: Desired event-action pairs

Now we give behavioral details.

Environment event	Desired action
E1 Time to download list of joiners	Personnel Information System knows that list of joiners is requested.
E2 Personnel Information System sends list of joiners.	Coordinator knows that list has been downloaded.
E3 System has been waiting for list of joiners too long.	Coordinator knows that list has not been downloaded.

- The SuD is not mentioned here.
- We describe desired behavior in the itshape environment.

Example 1: Required stimulus-response behavior

Stimulus	Current system state	Response	Next system state
Receive event "download list of joiners" from coordinator.	System contains no list of joiners.	send event "send me a list of joiners" to personnel information system.	System is waiting for list of joiners.
Receive list of joiners.	System is waiting for list of joiners.	Confirm to coordinator.	System contains list of joiners.
A timer times out.	System is waiting for list of joiners.	Inform coordinator of problem.	System contains no list of joiners.

Example 1: Which steps did we take?

- From desired system services we derived desired transactional event-action pairs in environment.
 - The service descriptions describe what is valuable to the environment.
 - The event-action pairs operationalize this as desired behavior in environment.
- From event-action pairs we derive stimulus-response pairs at the interface of the system.
 - To make the SR pairs transactional, a system state is introduced, which represents part of the environment state.

Compare our road map.

Example 2: Heating controller: Behavior to be enforced.

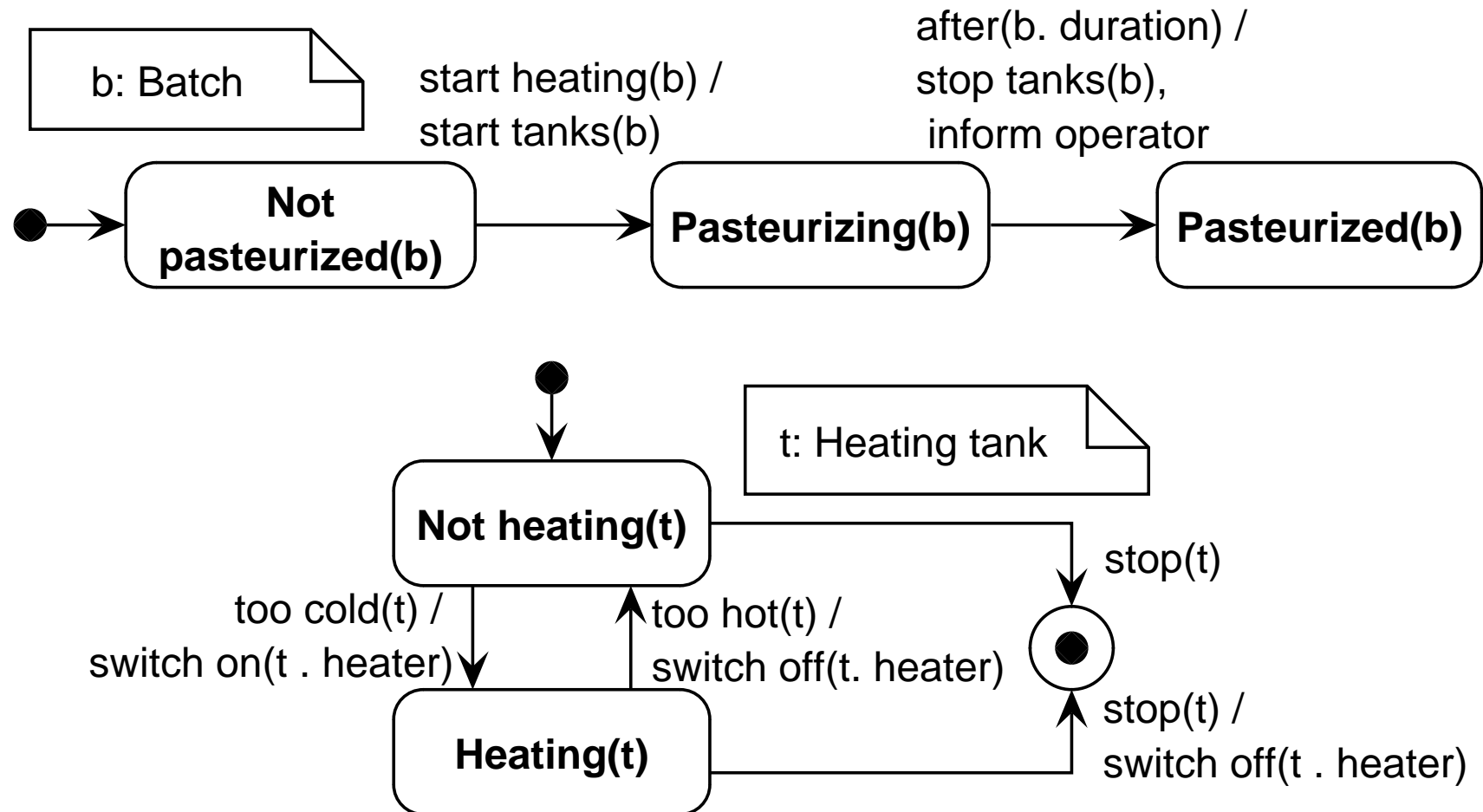
We start from a model of desired environment behavior.

Event	Desired action
Operator gives command to start heating batch b.	A heating process for the heating tanks of b is started. If at the start of the process, temperature in a tank is too low, the heater of that tank is switched on. When during the process, a tank becomes 5 degrees Celsius warmer than the desired temperature, its heater must be switched off. When it becomes 5 degrees Celsius colder than the desired temperature, its heater must be switched on. When the heating process has lasted for the duration of the recipe, heating must stop and the operator must be notified of this fact.

Example 2: Same behavior in atomic transitions

Event	Subject domain state	Desired action
E1 Operator gives command to start heating batch b		Heaters of tanks of b that are below recipe temperature, are switched on.
E2 Temperature in tank t rises 5 degrees above recipe temperature	The juice in t is being heated.	The heater of t is switched off.
E3 Temperature in tank t falls 5 degrees below recipe temperature	The juice in t is not being heated.	The heater of t is switched on.
E4 The heating duration has passed, counted since the start of heating of b.	b is being heated.	<ul style="list-style-type: none"> • Heaters of b that are on, are switched off. • Operator is informed.

Example 2: Same behavior described by two diagrams



Example 2: Definitions

- **start tanks(b: Batch)**. For each tank t of b , signal “start(t)”.
- **stop tanks(b: Batch)**. For each tank t of b , signal “stop(t)”.

These supplement the diagram.

Example 2: Stimulus-response behavior that enforces environment behavior

Stimulus	Current controller state	Desired response	Next state
S1 Operator gives command to start heating batch b	Not heating t and not heating b.	Switch on heaters of tanks with low temperature.	Heating t and b.
S2 Every 60 seconds.	Heating t and measured temp $>$ desired temp + 5	Controller switches off the heater of t.	Heating t.
	Heating t and measured temp $<$ desired temp - 5	Controller switches on heater of t.	Heating t.
S4 Recipe time since the start of heating of b has passed.	Heating b.	<ul style="list-style-type: none"> • Switch off heaters of b that are on. • Inform operator. 	Not heating b.

Example 2: Questions

The behavior brought about by the SR pairs is not exactly the same as the behavior that is desired.

- What are the differences?
- Why are these acceptable?

Example 2: Which steps did we take?

(Compare our roadmap)

- Desired environment behavior was modeled by means of non-atomic event-action pair.
- This was decomposed into atomic event-action pairs.
- From this we derived desired system behavior as a set of atomic stimulus-response pairs, that would approximately bring about desired environment behavior.

Part IV: Main points

- Behavior occurs in the environment and at all levels in the system.
- Find required system behavior by
 - Analyzing required system services or
 - modeling desired environment behavior.
- Behavior can be described in tabular format or by diagrams.

Structure of part IV

- State transition lists and tables
- State transition diagrams
- Execution semantics
- Modeling and design guidelines

Chapter 11. State Transition Lists and Tables

- Range from informal to formal
- Can be used at system level down to component or software object level.
- Different kinds of lists and tables:
 - Event list
 - Stimulus-response list
 - State transition table
 - Decision table

At system level, the lists are usually informal. At software object level, they are usually formal.

The use of behavior descriptions

Behavior descriptions can be used to represent

- Assumed environment behavior
- Desired environment behavior
- Required system behavior
- Required component behavior

You cannot tell how a behavior description is used by reading it.

Each behavior description must be supplemented by an indication of its intended use.

Event lists

List of event descriptions and, for each event, a description of its effect.

Example: description of assumed device behavior.

- **light on(b).** If b is not already in state On(b), then it enters state On(b). Otherwise, nothing changes.
- **light off(b).** If b is not already in state Off(b), then it enters state Off(b). Otherwise, nothing changes.

Example event list: desired subject domain behavior

Event	Desired action
Operator gives command to start heating batch b.	A heating process for the heating tanks of b is started. If at the start of the process, temperature in a tank is too low, the heater of that tank is switched on. When during the process, a tank becomes 5 degrees Celsius warmer than the desired temperature, its heater must be switched off. When it becomes 5 degrees Celsius colder than the desired temperature, its heater must be switched on. When the heating process has lasted for the duration of the recipe, heating must stop and the operator must be notified of this fact.

Example state transition table: Desired subject domain behavior

Event	Subject domain state	Desired action
E1 Operator gives command to start heating batch b		Heaters of tanks of b that are below recipe temperature, are switched on.
E2 Temperature in tank t rises 5 degrees above recipe temperature	The juice in t is being heated.	The heater of t is switched off.
E3 Temperature in tank t falls 5 degrees below recipe temperature	The juice in t is being heated.	The heater of t is switched on.
E4 The heating duration has passed, counted since the start of heating of b.	b is being heated.	<ul style="list-style-type: none"> • Heaters of b that are on, are switched off. • Operator is informed.

Effect descriptions

- **Transactional.** Description of one state transition.
 - Intermediate states abstracted away (i.e., atomic).
 - Passage of time abstracted away (instantaneous).
- **Scenario.**
 - Description of several state transitions, with intermediate states (i.e. not atomic).
 - Progress of time.

To transform a scenario description into a transactional list, we need to introduce states.

State transition tables (STTs)

List of transactional entries of the form (event, current state, actions, next state).

Variables are local to one entry. Bound in the left-hand side.

Stimulus	Current controller state	Controller response	Next controller state
pass doors(c)	Opened(c)		Opened(c)
	Closing(c)	open doors(c)	Opened(c)
10 seconds after the most recent execution of <code>c.state := Opened</code>	Opened(c)	close doors(c)	Closing(c)

Adding an initial state and initialization action

Initially		close doors(c)	Closing(c)
Stimulus	Current controller state	Controller response	Next controller state
pass doors(c)	Opened(c)		Opened(c)
	Closing(c)	open doors(c)	Opened(c)
10 seconds after the most recent execution of $c.state := \text{Opened}$	Opened(c)	close doors(c)	Closing(c)

Transformation table: no state change

Event	Current state	Action
arrive(b, c)	Destination request(b, c) and Atfloor(b, c)	<ul style="list-style-type: none"> • stop motor(b, c) • open doors(c) • light off(b)
	Forward request(b, c) and Atfloor(b, c)	<ul style="list-style-type: none"> • stop motor(b, c) • open doors(c) • light off(b) • show direction(c)
	Outermost reverse request(b, c) and At- floor(b, c)	<ul style="list-style-type: none"> • stop motor(b, c) • open doors(c) • light off(b) • reverse and show di- rection(c)

Transformation table = Decision table

bd, bf, br: Request button c: Elevator cage					
Destination request(bd, c) and Atfloor(bd, c)	T	F	T	F	T
Forward request(bf, c) and Atfloor(bf, c)	F	T	T	F	F
Outermost reverse request(br, c) and Atfloor(br, c)	F	-	-	T	T
stop motor(c)	X	X	X	X	X
open doors(c)	X	X	X	X	X
show direction(c)	X	X	X		
reverse and show direction(c)				X	X
light off(bd)	X		X		X
light off(bf)		X	X		
light off(br)				X	X

An STT with next states but without actions

Current state	button	Event	Next button state
On(b)		light off(b)	Off(b)
		light on(b)	On(b)
Off(b)		light on(b)	On(b)
		light off(b)	Off(b)

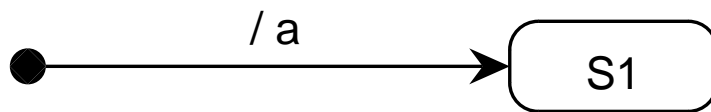
Main points

- Tabular behavior descriptions range from informal (event list) to formal (state transition table).
- Can be used for assumed or desired behavior of environment or system.
- List entry can describe a transition or a scenario.
- Variables can be declared for a table; binding is local to one entry.
- Stateless “transition” is really decision rule.

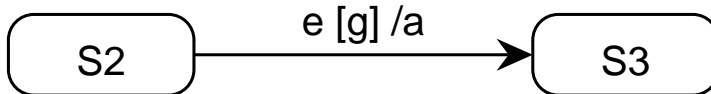
Chapter 12. State Transition Diagrams

- Good at showing the structure of behavior.
- Not good at showing unstructured behavior.
- Can only show a limited amount of information because restricted to one sheet of paper.
- Tables and lists on the other hand can show unlimited amounts of information, because they can continue on any number of sheets.

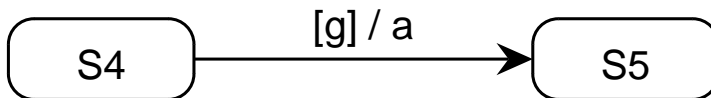
Mealy diagram constructs



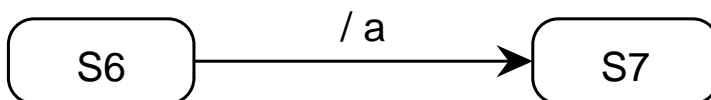
Initialization actions a, initial state S1.



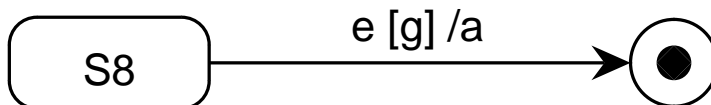
Trigger e, guard g, actions a.



Take transition when g becomes true, or immediately.

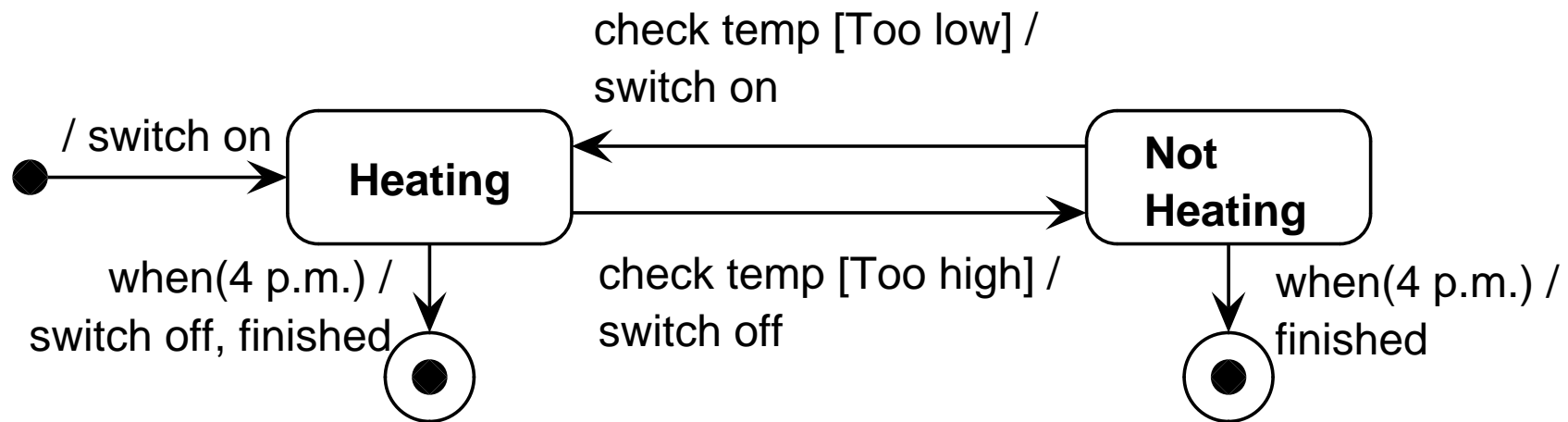


Take transition immediately.



Transition to final state.

Example

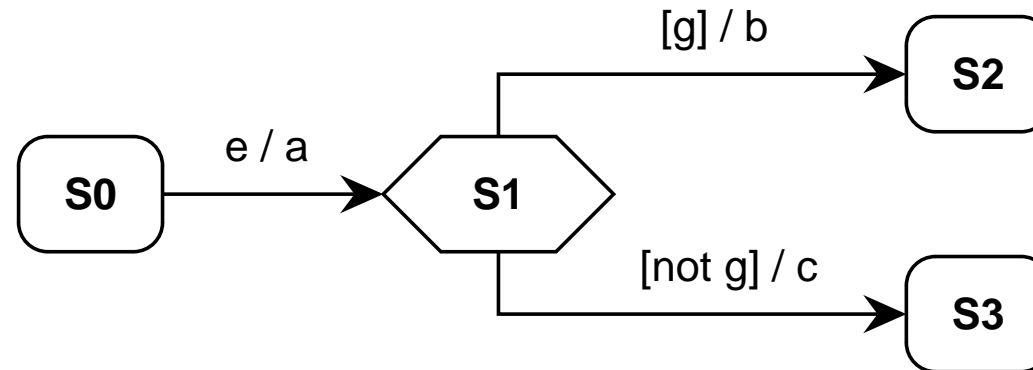


Events

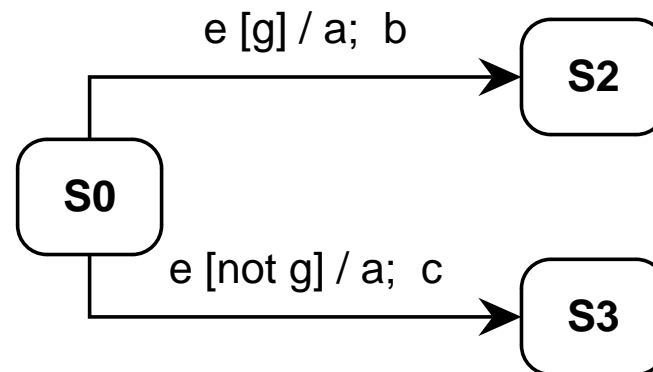
An **event** is a discrete change in the condition of the world.

- **Named events.** We gave a name to it.
- **Condition change event.** A Boolean condition g becomes true.
- **Temporal event.** Significant moment in time. Relative temporal event: $\text{after}(t)$, absolute temporal event: $\text{when}(t)$.

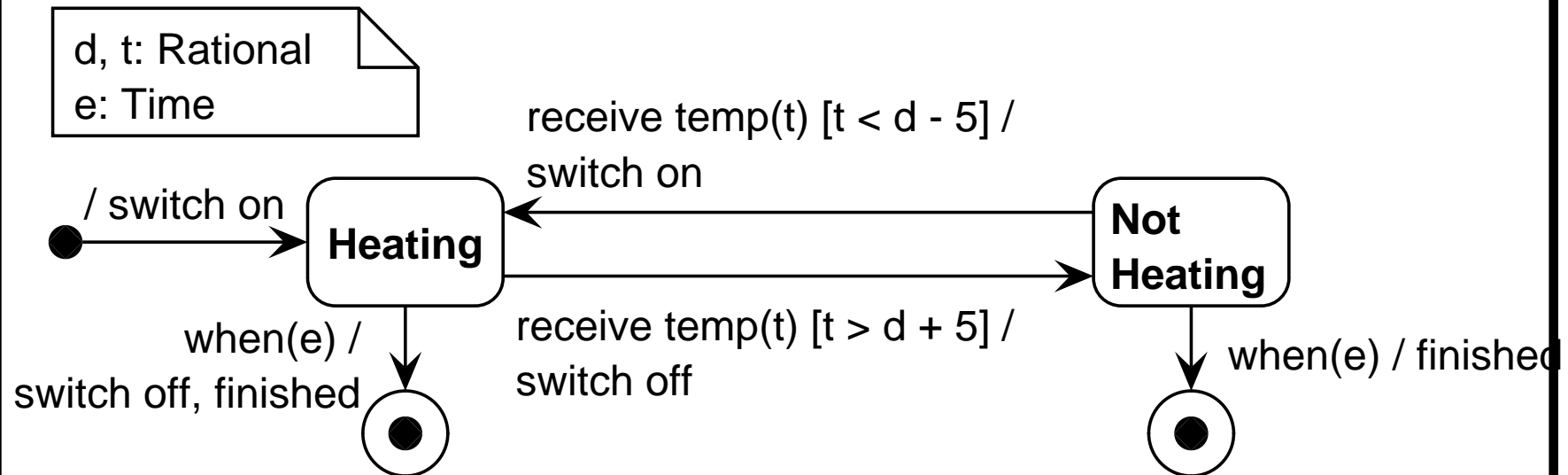
Decision states



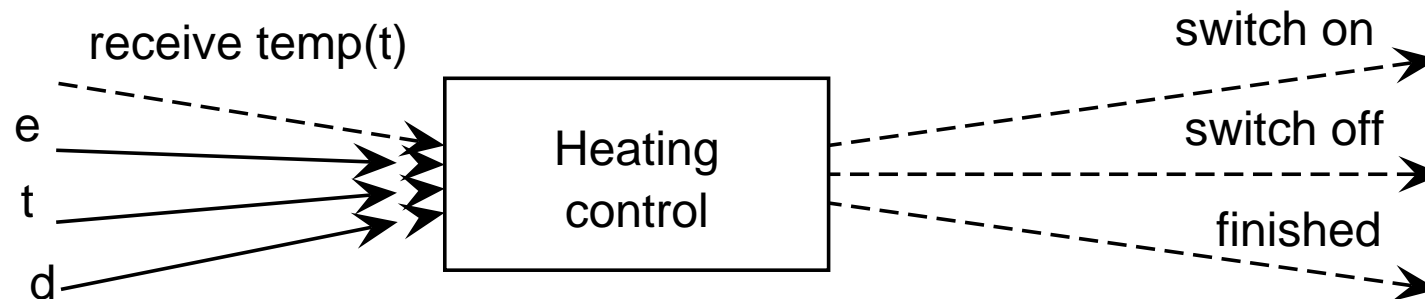
If g is not affected by a , this can be replaced by:



Variables

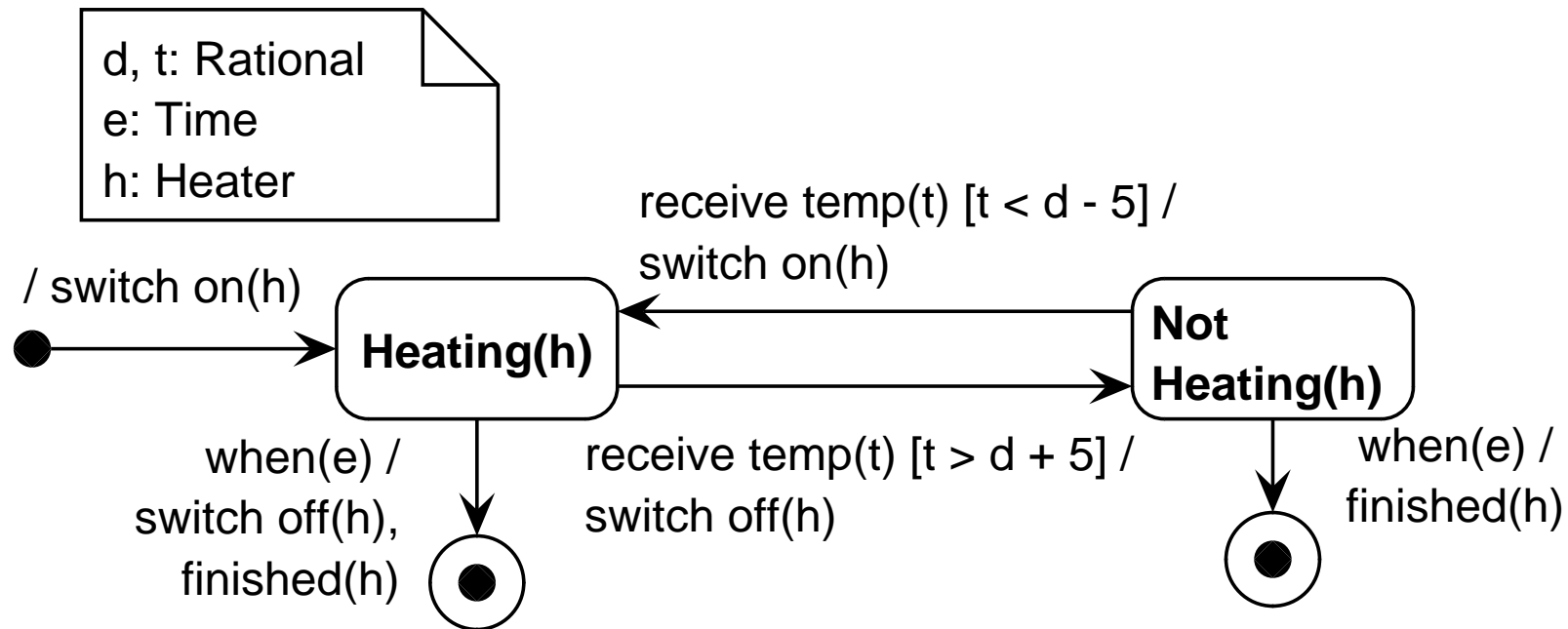


Interface of this machine:



Locality

Variables are local to a transition, except the identifier variable of a diagram.

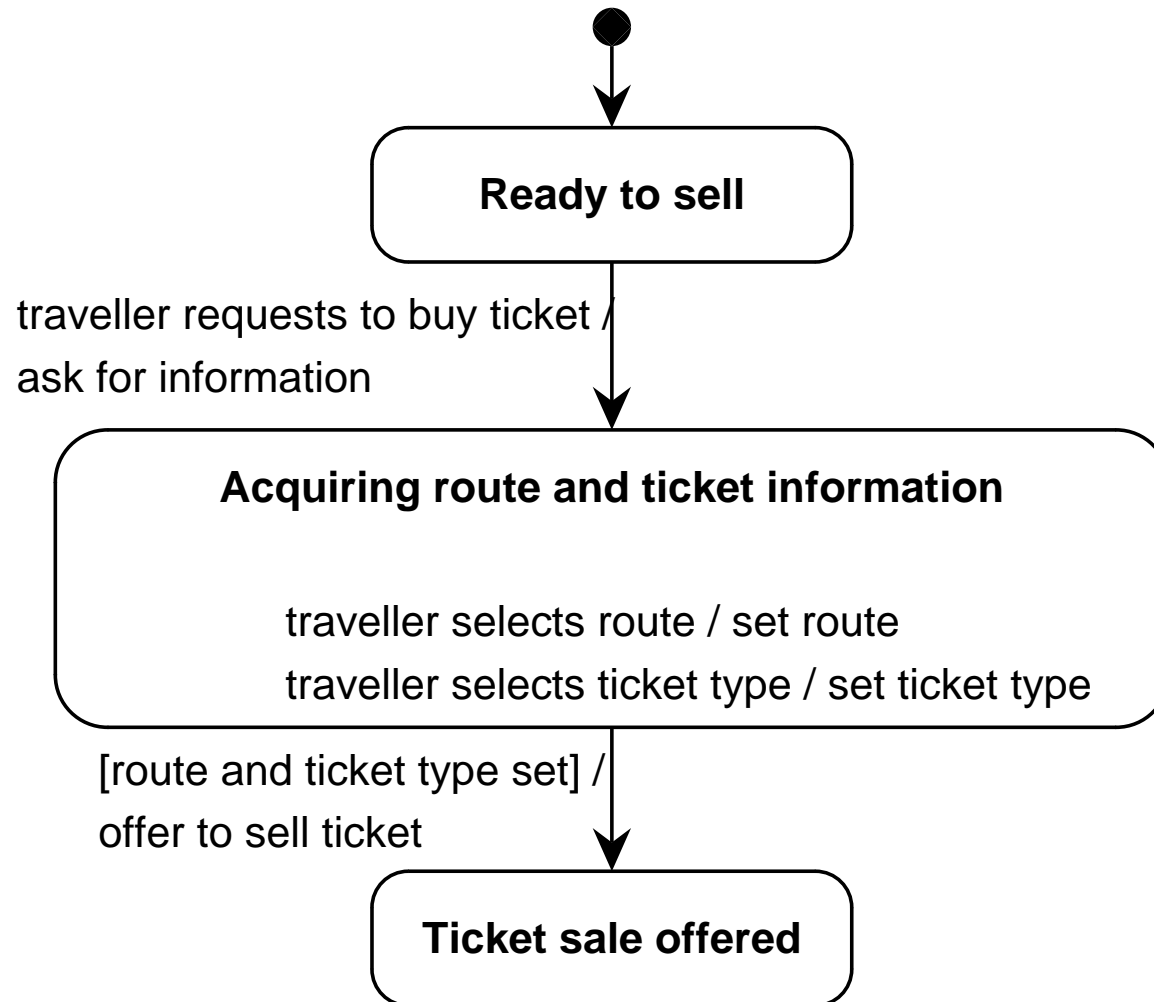


Statecharts

Mealy diagrams plus

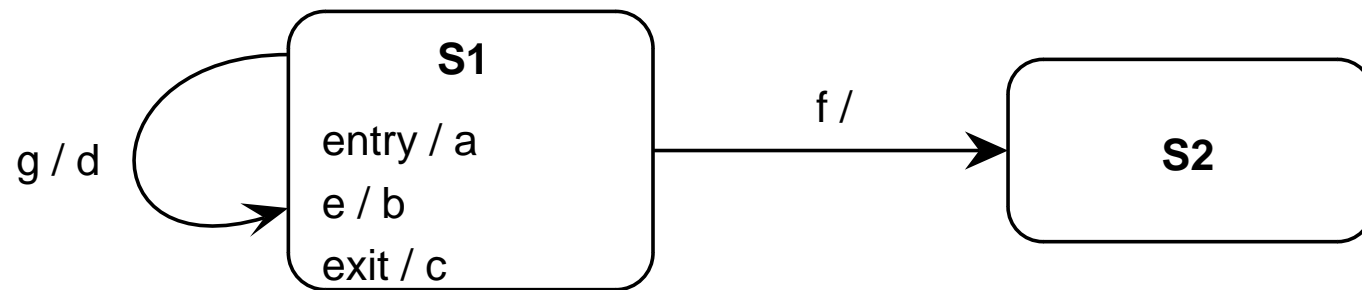
- State reactions
- State hierarchy
- Parallelism

State reactions



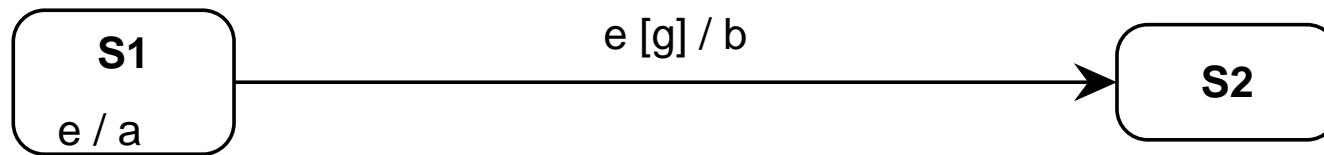
Saves space in diagram.

Example state reactions



What happens when g occurs?

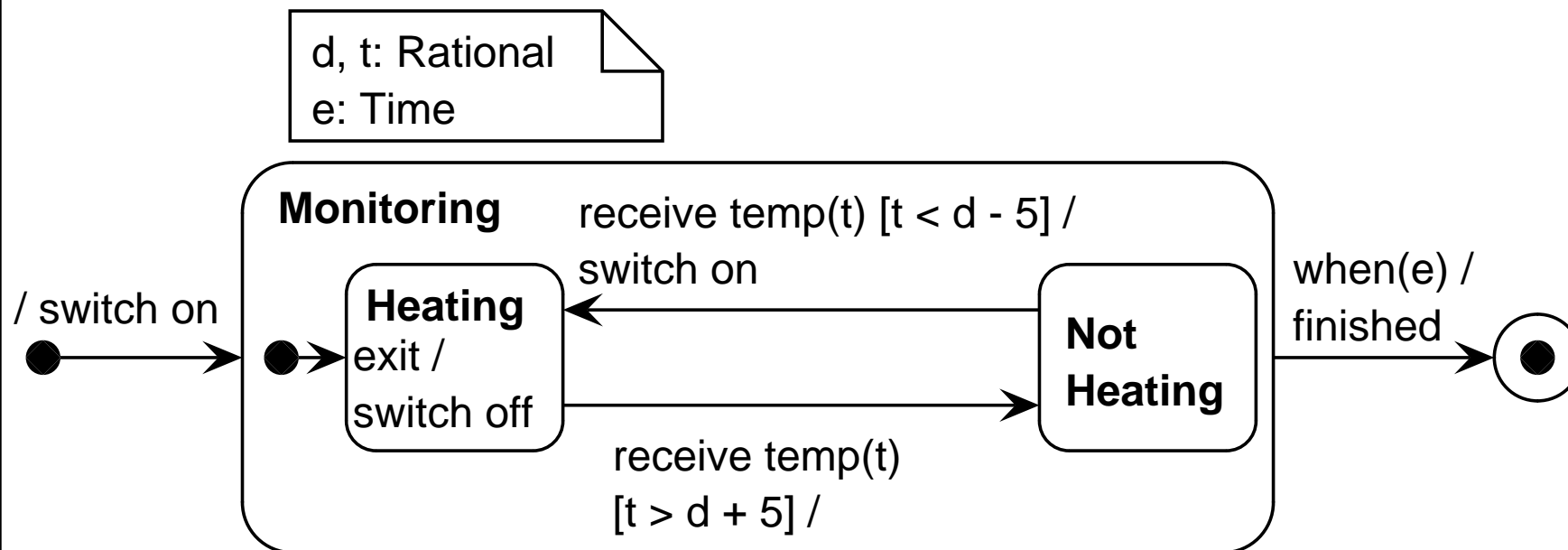
Conflict between state reaction and transition



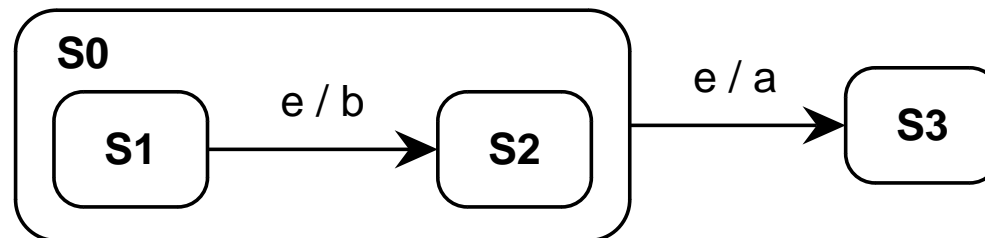
What happens when e occurs and g is true?

- Depends upon the semantics given to the notation by the analyst.
- All readers and authors of the diagram should use the same semantics!

Hierarchy



Conflicts between transitions at different levels

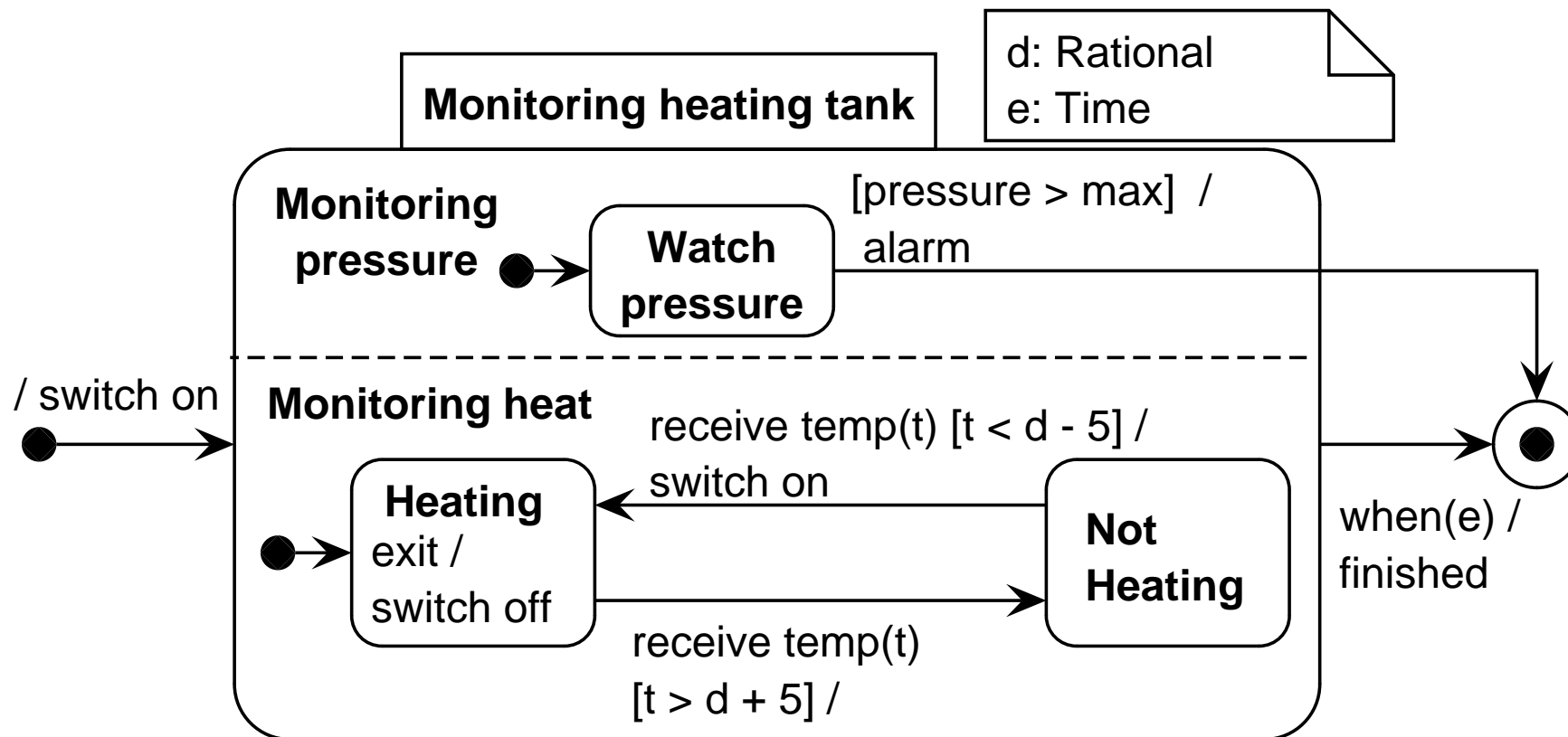


What happens when **e** occurs?

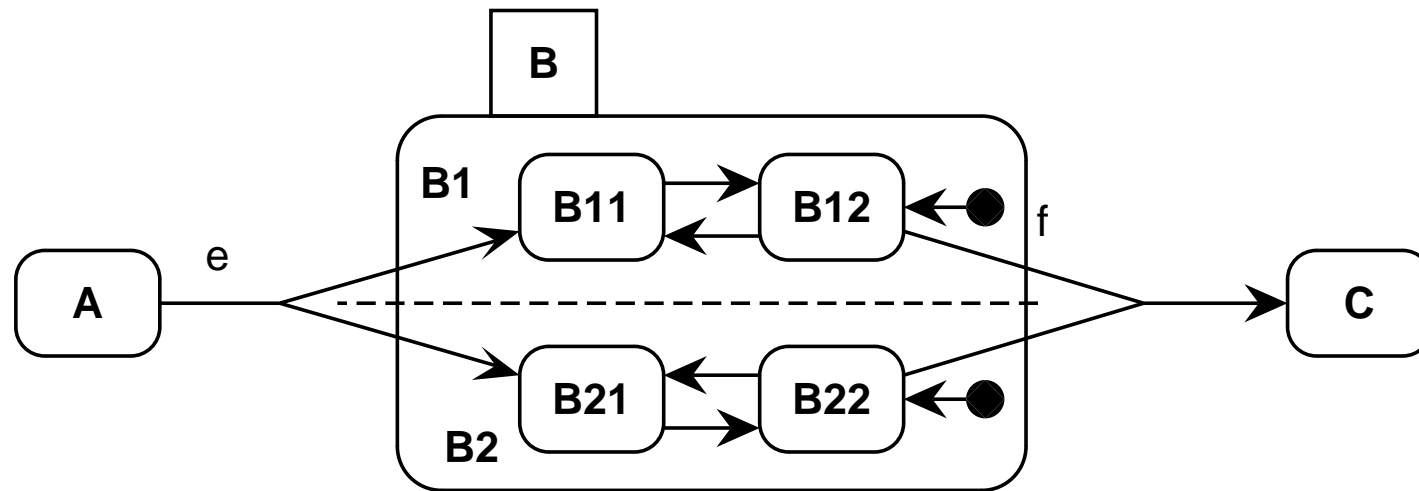
- UML: The lower transition is taken.
- Statemate: The higher transition is taken.

UML views lower-level behavior as specialization. Statemate sees higher-level behavior as more important, used e.g. to describe interrupts and error-handling.

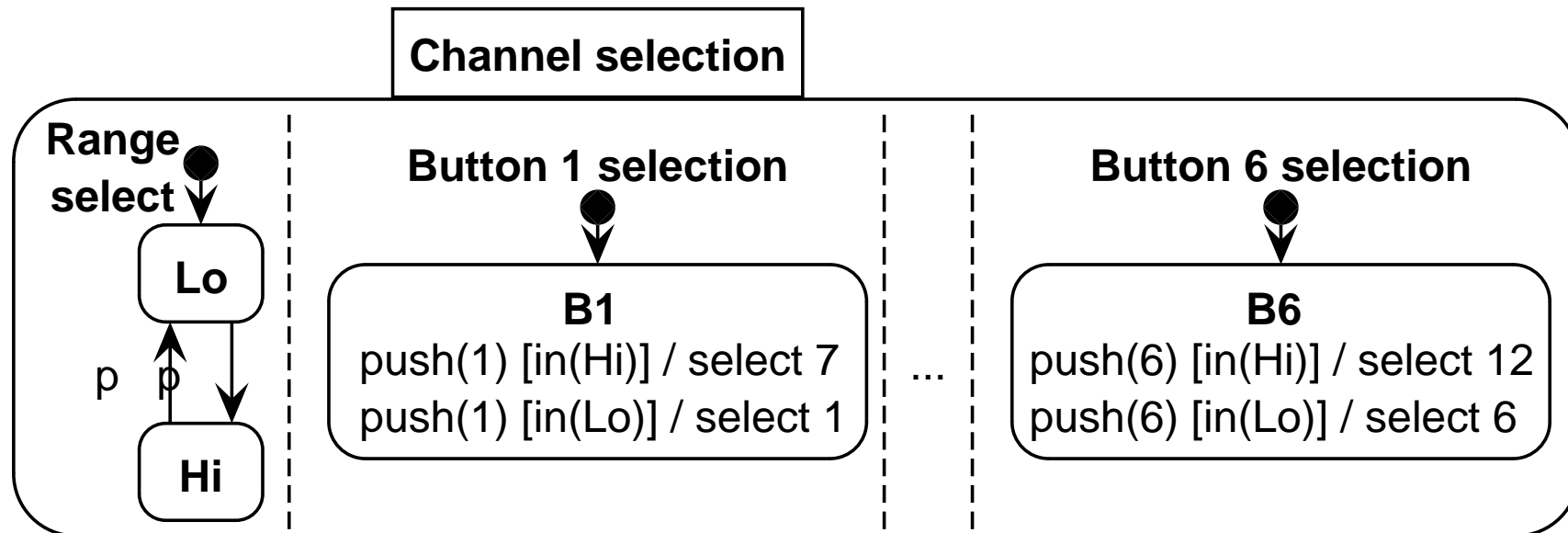
Parallelism



Hyperedges



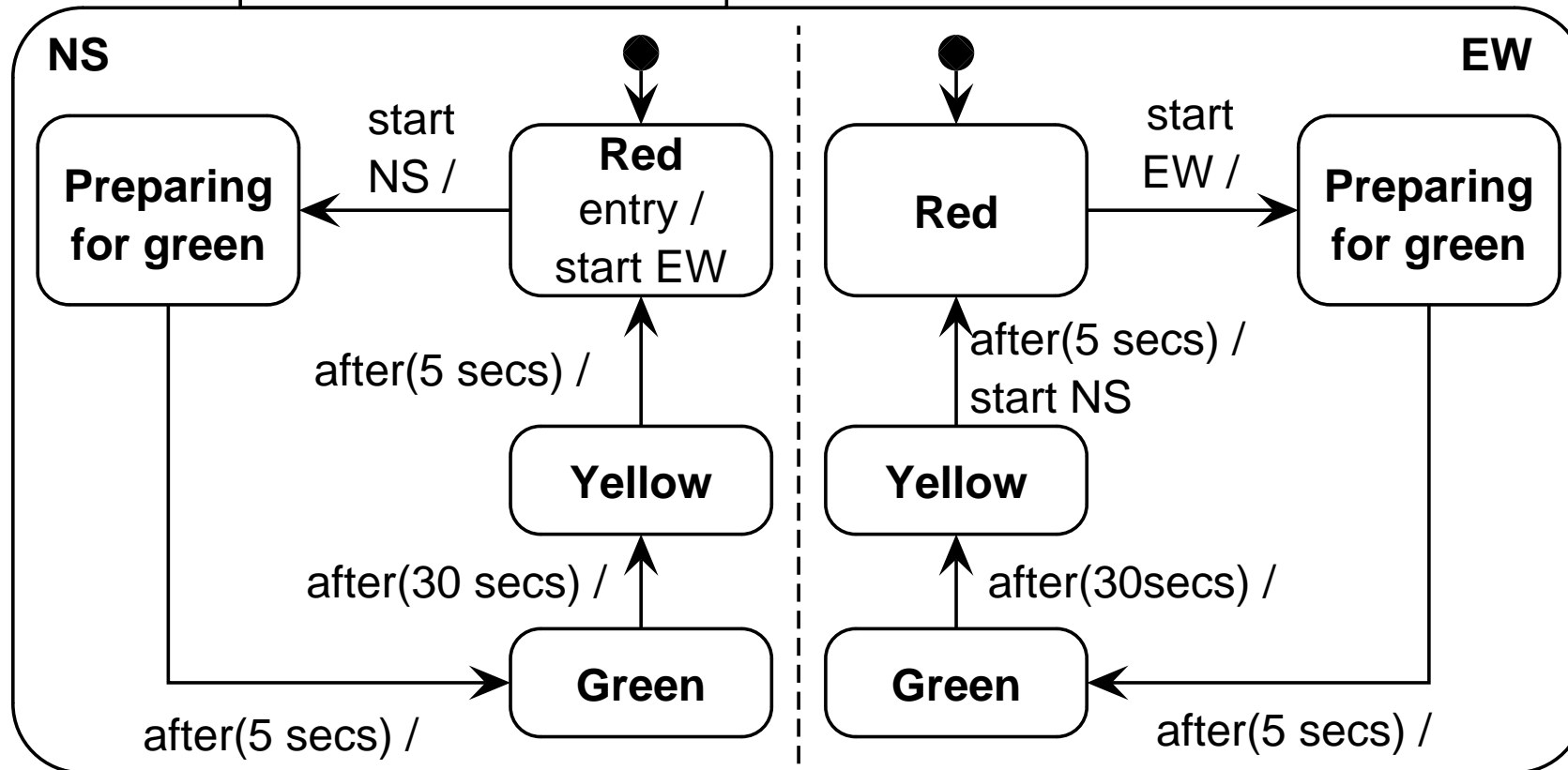
Breaking orthogonality by the $\text{in}(\text{State})$ predicate



Now the behavior in one state depends upon the state of some parallel component.

Breaking orthogonality by event broadcasting

Traffic light behavior

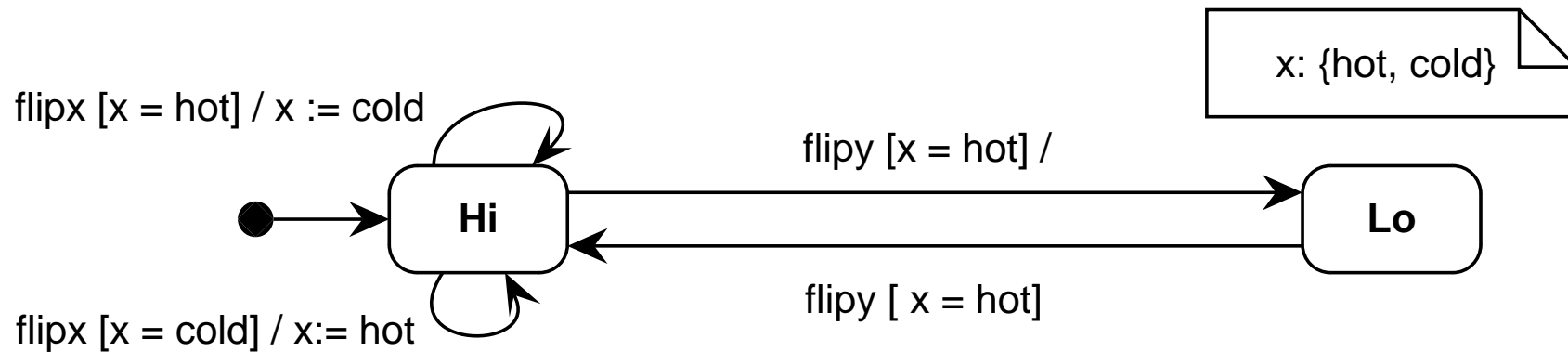


Now the behavior of one state depends upon events generated in some parallel component.

Main points

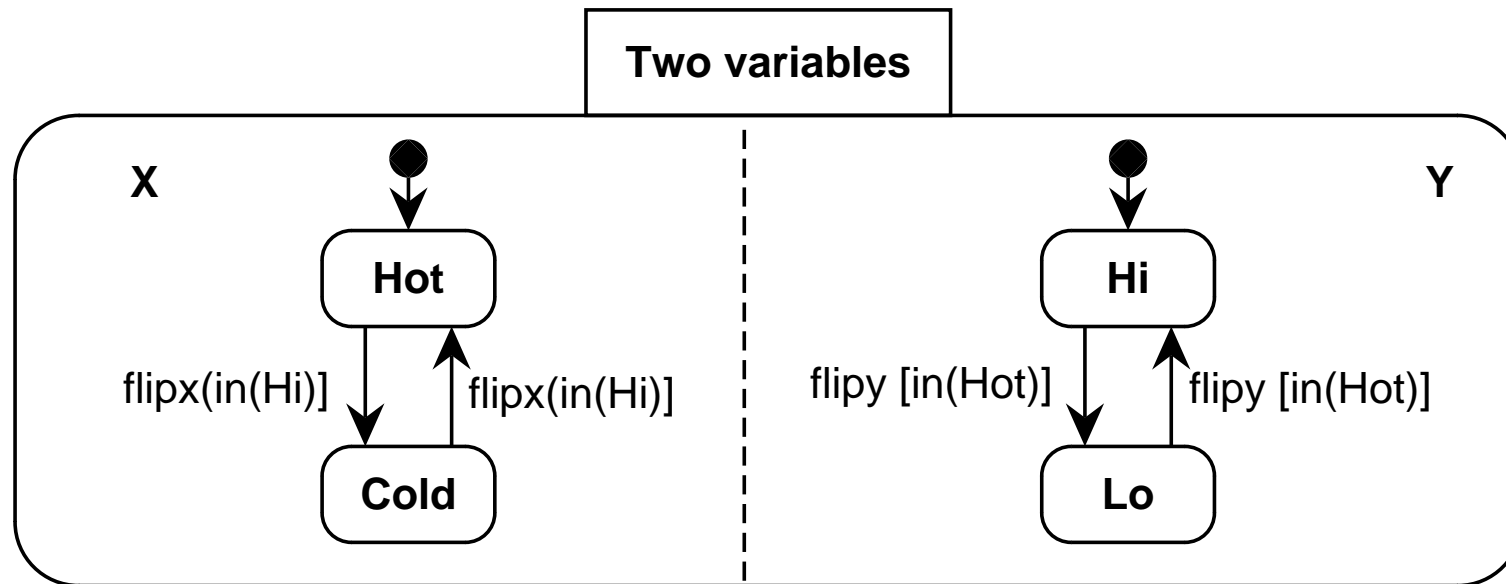
- STDs graphically represent reactive behavior, which consists of a discrete state space and $e [g] / a$ transitions.
- Events can be named, condition change, or temporal.
- Decision states are unstable.
- STDs can have local variables. Only the identifier variable is global to the diagram.
- Statecharts extend Mealy diagrams with state reactions, state hierarchy, parallelism.
- The semantics of a diagram must be defined explicitly (see chapter 13).

Questions



- Eliminate the variables as follows: Define two parallel components, one component for each variable.
- Is the resulting behavior equivalent to this one?

Solution

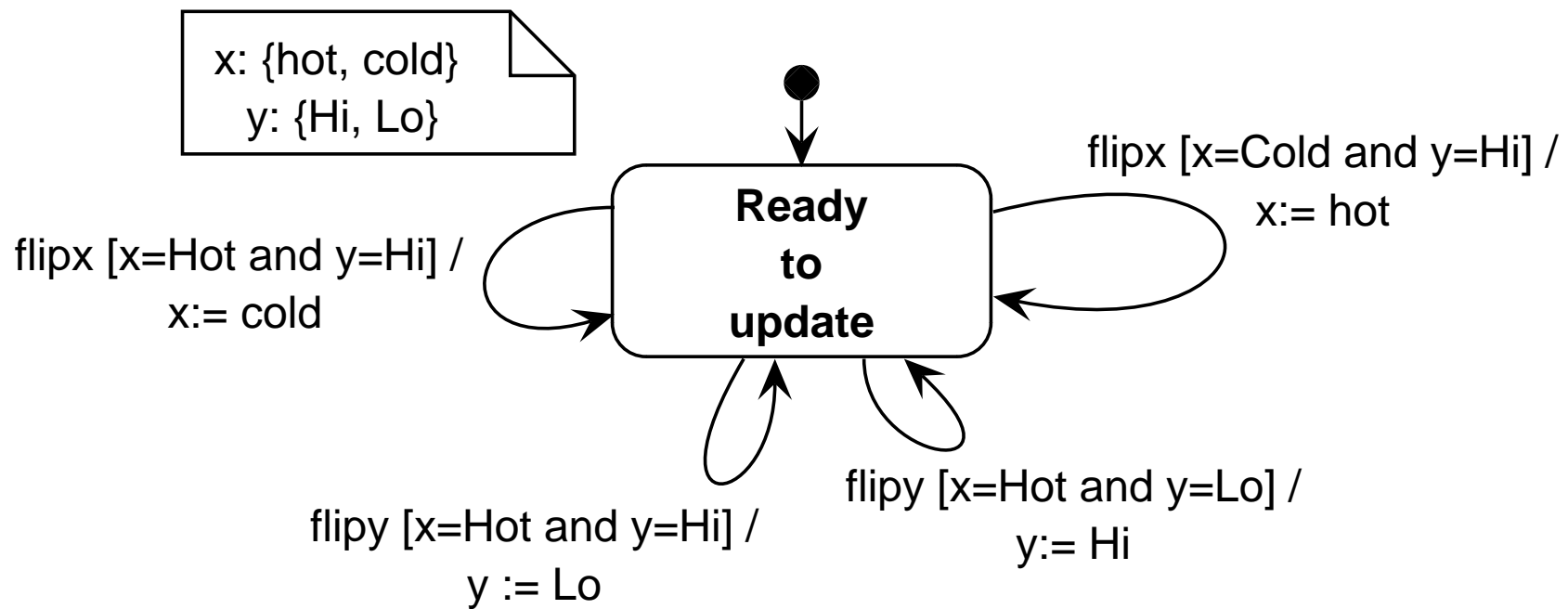


The important semantic difference between this statechart and the Mealy diagram is that the statechart can respond to a `flipx` and a `flipy` event at the same time.

Question

- Encode all state information in variables. The result should be an STD with one state only.
- Is the result equivalent to the original STD?

Solution



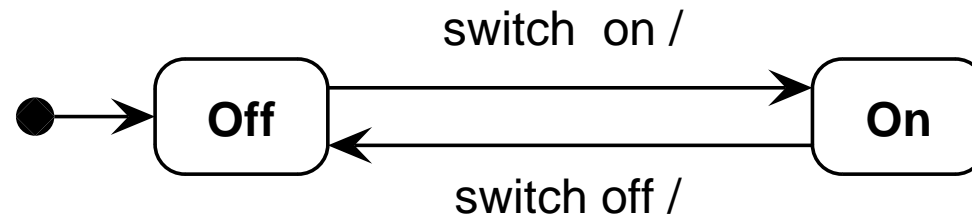
This diagram does not indicate the initial value of the variables.

Chapter 13. Behavioral Semantics

- STTs and STDs are descriptions of behavior.
- But they can be interpreted in many ways! That is a problem:
 - Designers, builders and users think they agree ...
 - ... but nevertheless expect different behavior to be implemented.
 - To execute a specification, it must be unambiguous.
 - To generate code, different code generation tools must use the same semantics.

In this chapter we review the relevant semantic choices.

Discretization

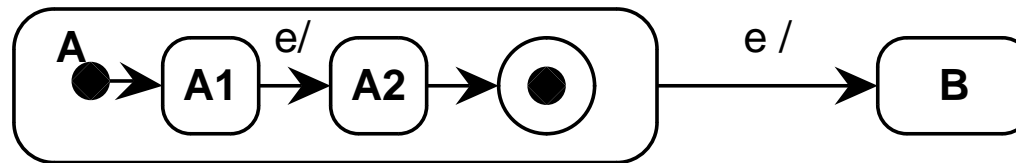


STDs and STTs are abstract, and therefore necessarily incomplete, descriptions of continuous behavior.

- **Atomicity.** Ignore intermediate states of each transition.
- **Possibility.** There are no slightly different events or states.
switch on is possible when Off etc.
- **Isolation.** Whatever else happens, switch on in state Off leads to state On, etc.
- **Durability.** A state is stable until an event occurs that triggers an outgoing transition.

Wait States and Activity States: Semantic options

An activity state has internal activity, a wait state has not.



This creates the possibility for conflict: What happens when *e* occurs? Options:

- **Deferred response.**
- **Forced termination.**
 - Higher level has priority (Statemate).
 - Lower level has priority (UML).

Final states: Semantic options

Activity states create ambiguity for final state node. What happens when final state is reached?

- **Global termination:** bull's eye indicates that overall process terminates (Statemate).
- **Local termination:** Activity terminates; wait until e occurs (UML).

Pre- and postconditions: Problems

	Current state	Event	Action	Next state
1	$P(x)$	$e(x, y)$	$a(x, y)$	$Q(x)$

- Does $P(x)$ become false?
- Should $Q(x)$ be false in initial state?
- We will treat state descriptions in an STT as **preconditions** and **postconditions**, that constrain the current and next state. So:
 - $Q(x)$ can have any value in initial state.
 - $P(x)$ can have any value in next state.

If you don't want this, write down truth value explicitly.

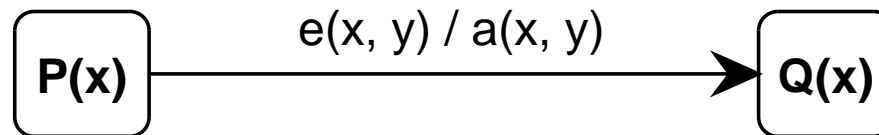
Pre- and postconditions: Possible solutions

	Current state	Event	Action	Next state
1	$P(x)$	$e(x, y)$	$a(x, y)$	$Q(x)$
2	$P(x)$	$f(x, y)$	$a(x, y), b(y)$	$R(x)$

- What happens with $R(x)$ in transition 1? Not specified.
- If we must write down truth values explicitly, table entries become very large.
- **Derivation rules.** E.g. $Q(x) \rightarrow R(x)$. Part of domain knowledge or SuD spec.
- **Frame rule.** If the transition rule and derivation rules do not imply that P changes, then it does not change.

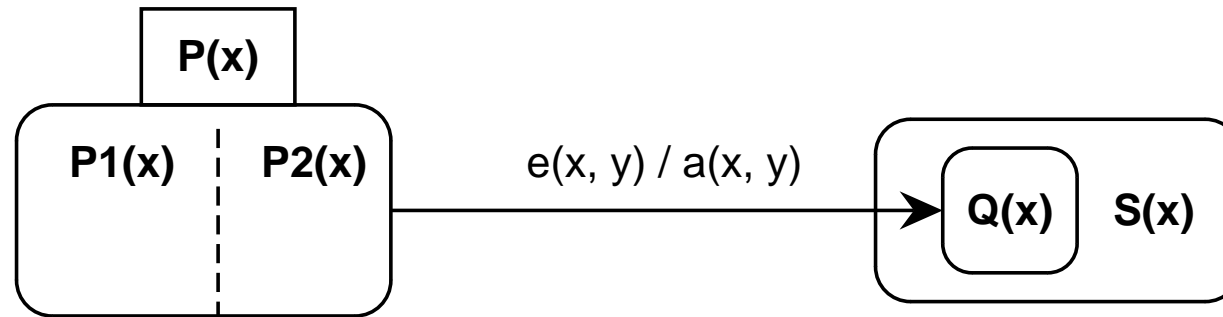
Pre- and postconditions in STTs and STDs

	Current state	Event	Action	Next state
1	$P(x)$	$e(x, y)$	$a(x, y)$	$Q(x)$



- Equivalent?
- No: STDs come with the interpretation rule that $S_i \leftrightarrow \neg S_j$ for $i \neq j$.
- So $P(x) \leftrightarrow \neg Q(x)$ in the STD!

Interpretation rules for statecharts

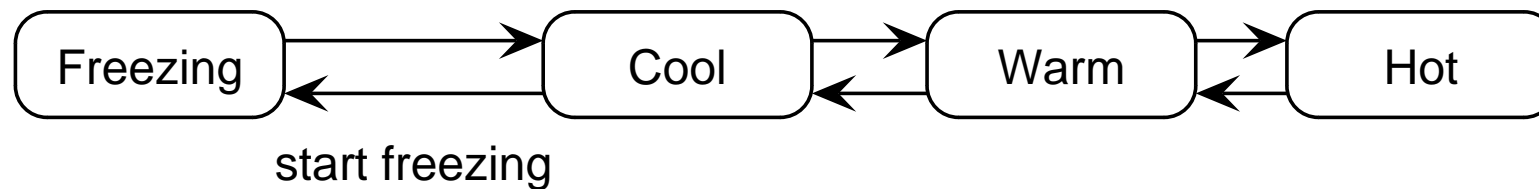


Use the following interpretation rules for statecharts:

- For all x , $P(x) \leftrightarrow \text{not } S(x)$.
- For all x , $Q(x) \rightarrow S(x)$.
- For all x , $P1(x) \rightarrow P(x)$.
- For all x , $P2(x) \rightarrow P(x)$.

Triggering (1)

The first problem with triggering is how we interpret what the diagram does *not* say.



Can start freezing occur in state Hot?

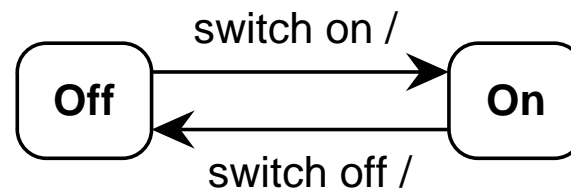
- **Physically impossible:** Contradicts the laws of nature.

If this is what we mean, we should add this interpretation rule to the diagram.

Triggering (2)

Definition:

- **switch on** is the event that leads the system from Off to On.



What does this mean? Can switch on occur in state On?

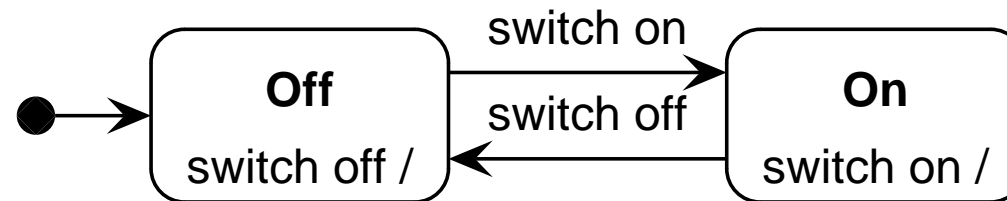
- **Logically impossible:** Contradicts the definition in the dictionary. Whatever happens in state On is not the switch on event.
- **Logically possible:** switch on is a certain physical event that can occur in many states. Update the dictionary with this.

Whatever we mean, add this to the diagram.

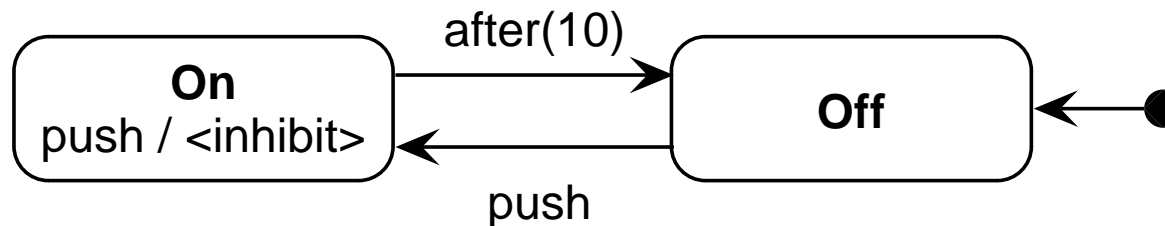
Triggering (3)

What happens if switch on *does* occur in state On?

- **Ignore:** Respond as if the event did not occur.

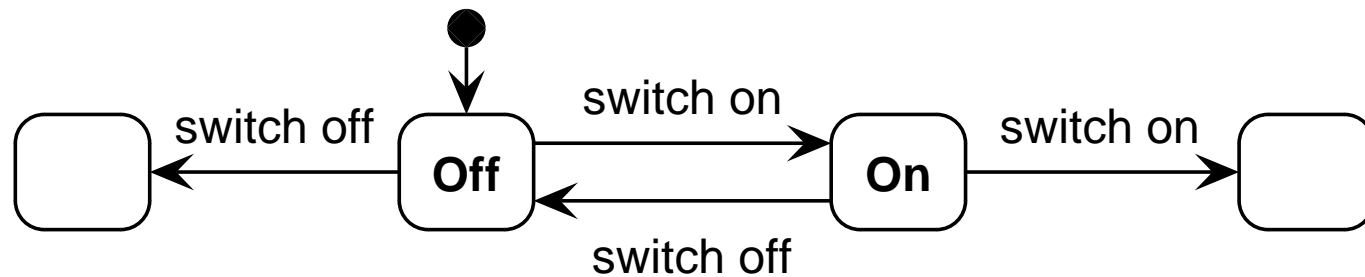


- **Inhibit:** Make the event physically impossible.



Triggering (4)

- **Unknown effect:** The system breaks down. This is the fragile semantics.



Make clear whether you agree on the ignore, inhibit, or fragile semantics as default interpretation for the STDs and STTs in a specification.

Triggering multiple transitions: The problem

	Current state	Event	Action	Next state
1	P(x)	e(x, y)	a(x, y)	Q(x)
4	R(y)	e(y, z)	b(y)	R(z)
5	R(x)	e(x, y)	c(x)	P(x)and not Q(x)

Suppose e(2, 4) occurs. Entries are instantiated as:

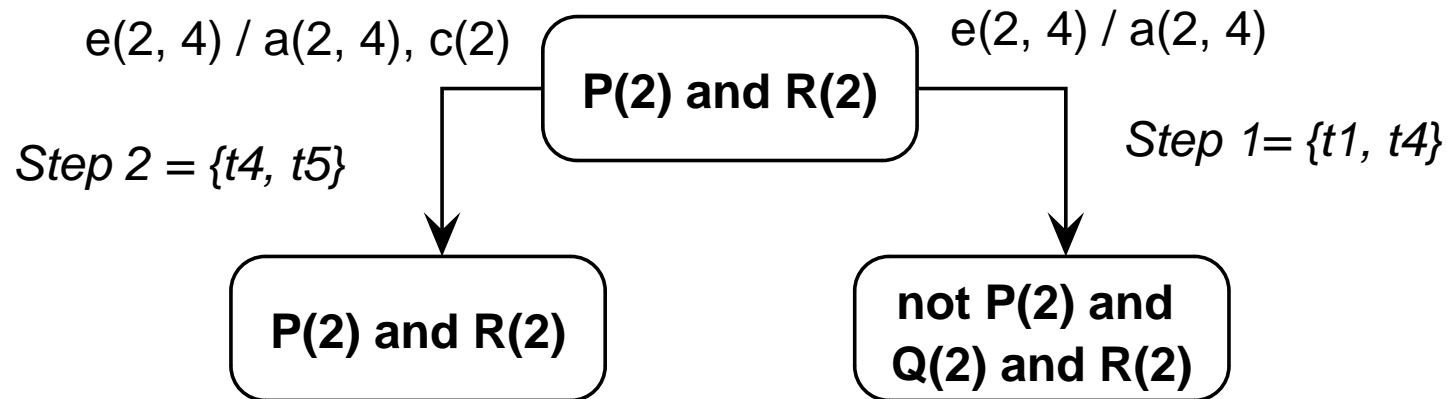
t1	P(2)	e(2, 4)	a(2, 4)	Q(2)
t4	R(2)	e(2, 4)	b(4)	R(4)
t5	R(2)	e(2, 4)	c(2)	P(2) and not Q(2)

Suppose P(2) and R(2) are true. What happens?

Triggering multiple transitions: Possible solutions

- **Step semantics.** Execute as many transitions as possible in one step. (Statemate and UML multi-threaded semantics)

Two possible steps:



- **Single-transition semantics.** Choose one transition non-deterministically. Forget about the others. (UML single-threaded semantics)

Responding to multiple events: The problem and possible solutions

Suppose $e(3, 4)$ and $f(3, 4)$ occur at the same time. What happens?

t1	P(3)	$e(3, 4)$	$a(3, 4)$	Q(3)
t2	P(3)	$f(3, 4)$	$a(3, 4)$	R(3)

- **Concurrent-event semantics.** Respond to all events not yet responded to. (Statemate)
- **Sequential-event semantics.** Respond to events in some sequence. (UML)

Multistep semantics: The problem

One transition can trigger further transitions. E.g. suppose $f(3, 4)$ happens in state $P(3)$:

t2	$P(3)$	$f(3, 4)$	$a(3, 4), b(4)$	$R(3)$
----	--------	-----------	-----------------	--------

Action $a(3, 4)$ in turn triggers an instance of entry 3:

t3	$R(3)$	$a(3, 4)$	$b(3)$	$P(3)$ and not $R(3)$
----	--------	-----------	--------	--------------------------

What happens?

Multistep semantics: Possible solutions

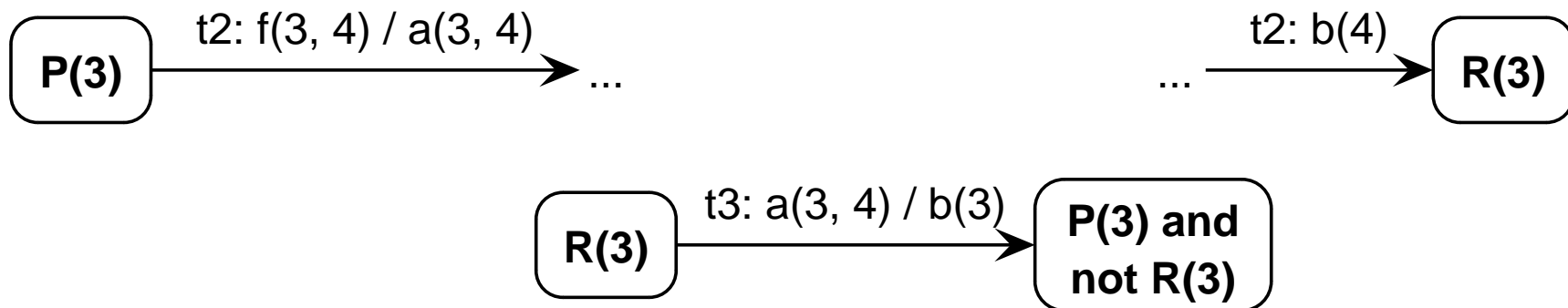
- **Single step semantics.** After each step, the behavior can respond to external as well as internally generated events. (Statemate option)
- **Superstep semantics.** The behavior first performs the complete multistep response before it is able to respond to external events. (Statemate option)
- **Delayed step semantics.** The behavior responds to internal and external events in some later step. (UML)

Action semantics

How is a complex action such as a, b, c executed?

- **Concurrent action semantics.** The actions in a complex action s are all executed simultaneously. (Statemate)
- **Sequential action semantics.** The actions are executed in some sequence. (UML)

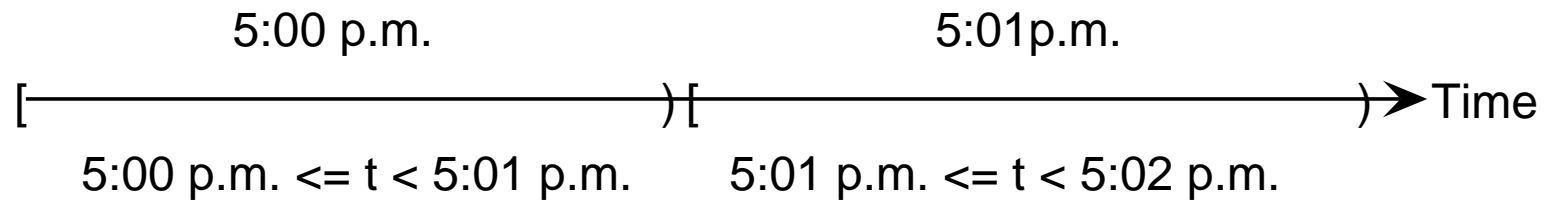
What if a UML action is an operation call?



Time

A “**time point**” is a time interval whose length is not significant for the description.

Abstract time points

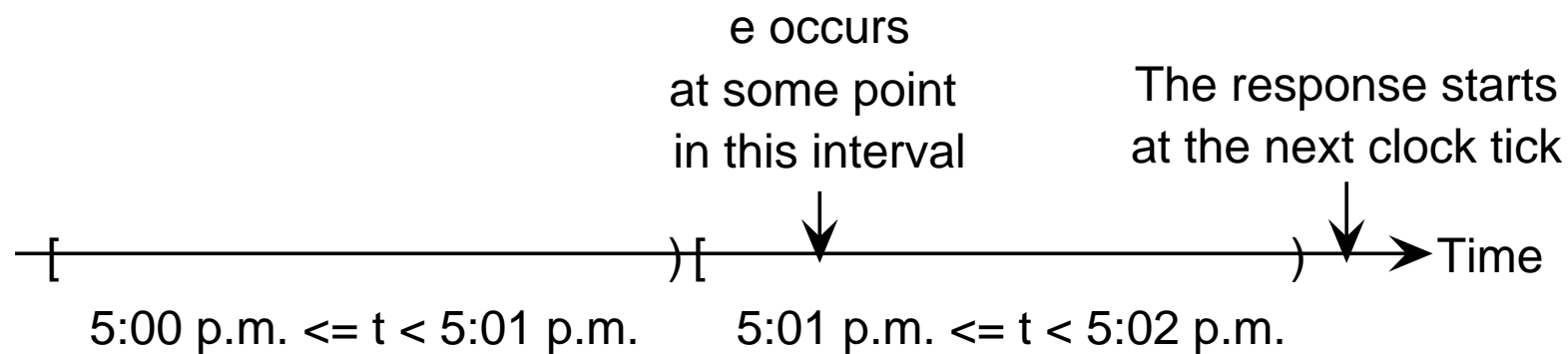


Real time

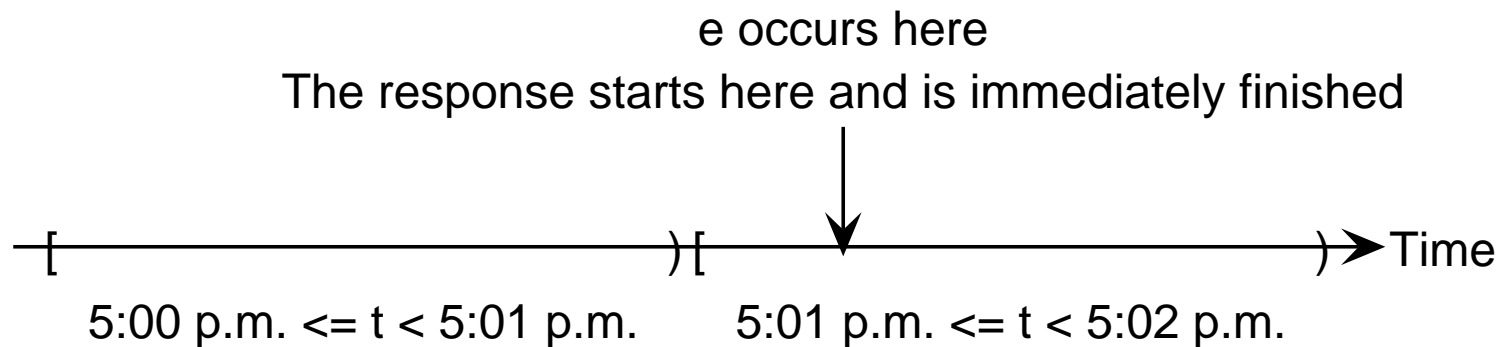
Events

Our events are abstractions: They are instantaneous.

- **Clock-driven semantics** (Statemate)



- **Event-driven semantics** (Statemate and UML)



Perfect technology assumption

No implementation restrictions. Infinite speed, unlimited memory.

- Clock-asynchronous: Respond immediately to an event occurrence.
- Perfect technology: Response takes no time to compute.

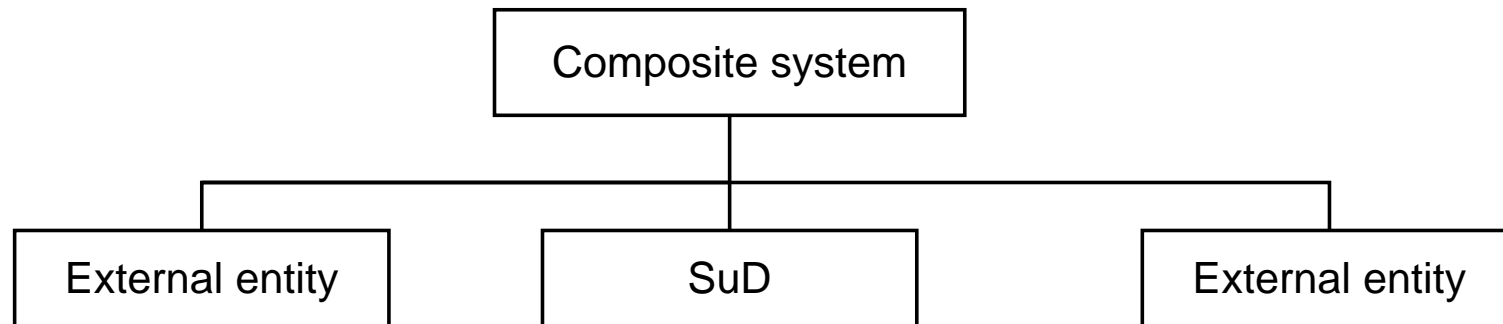
Together this implies a superstep semantics.

This agrees well with requirements-level models.

Main points

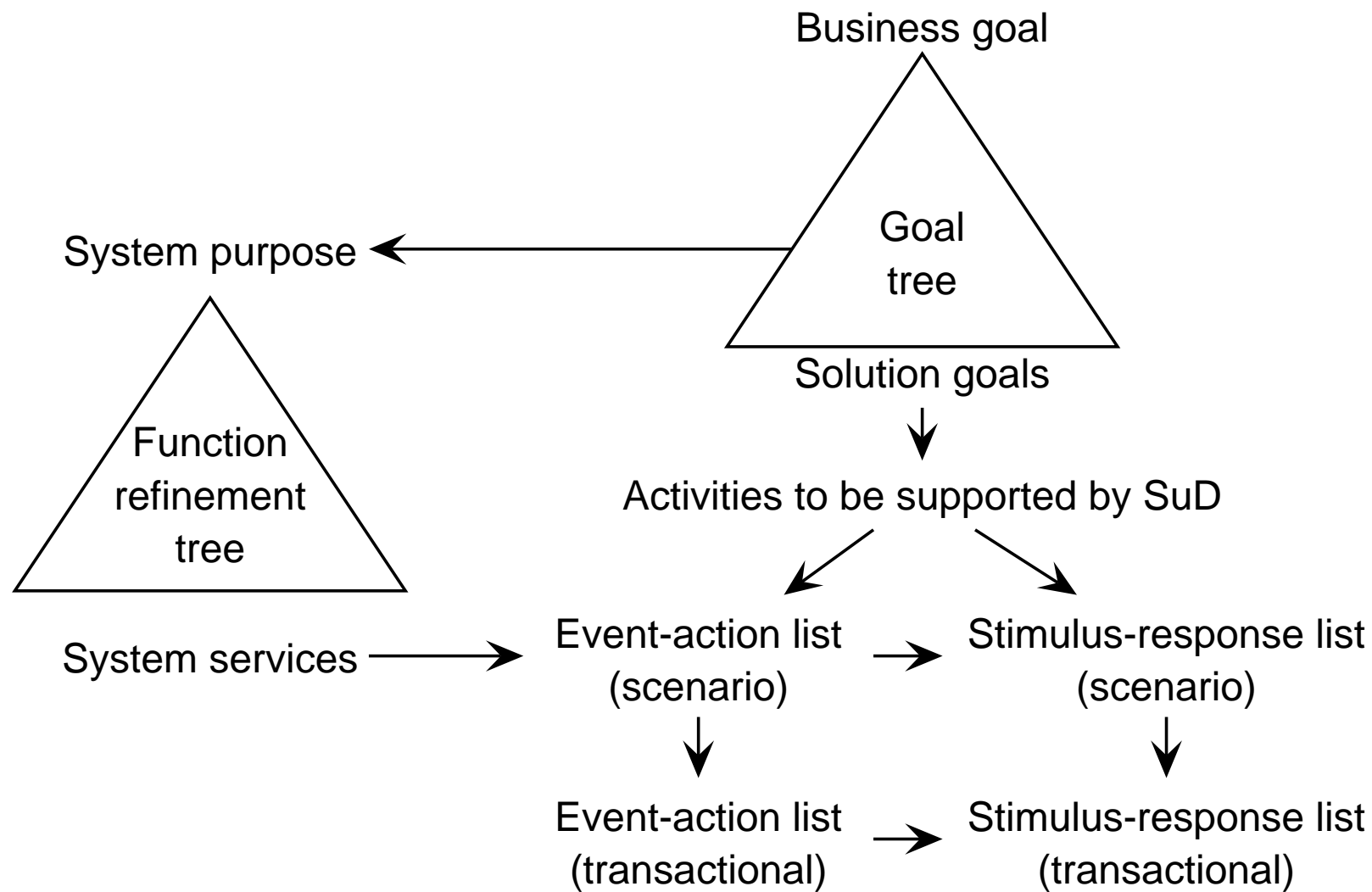
- STTs and STDs are describe discrete behavior that abstracts from continuous behavior in the real world.
- Conflict between transitions inside and out of activity states can be resolved differently.
- Final states can be global or local.
- Ambiguity in pre- and postconditions in STTs and STDs can be reduced by means of derivation rules and frame rules.
- The absence of event triggers can be interpreted in different ways: impossible, ignore, inhibit, unknown.
- Triggering multiple transitions: Steps versus single transitions.
- Derived transitions: Single steps, supersteps, delayed steps.
- Action semantics: Concurrent or sequential action execution.
- Time: Clock-driven or event-driven semantics. Perfect technology assumption.

Chapter 14. Behavior Modeling and Design Guidelines

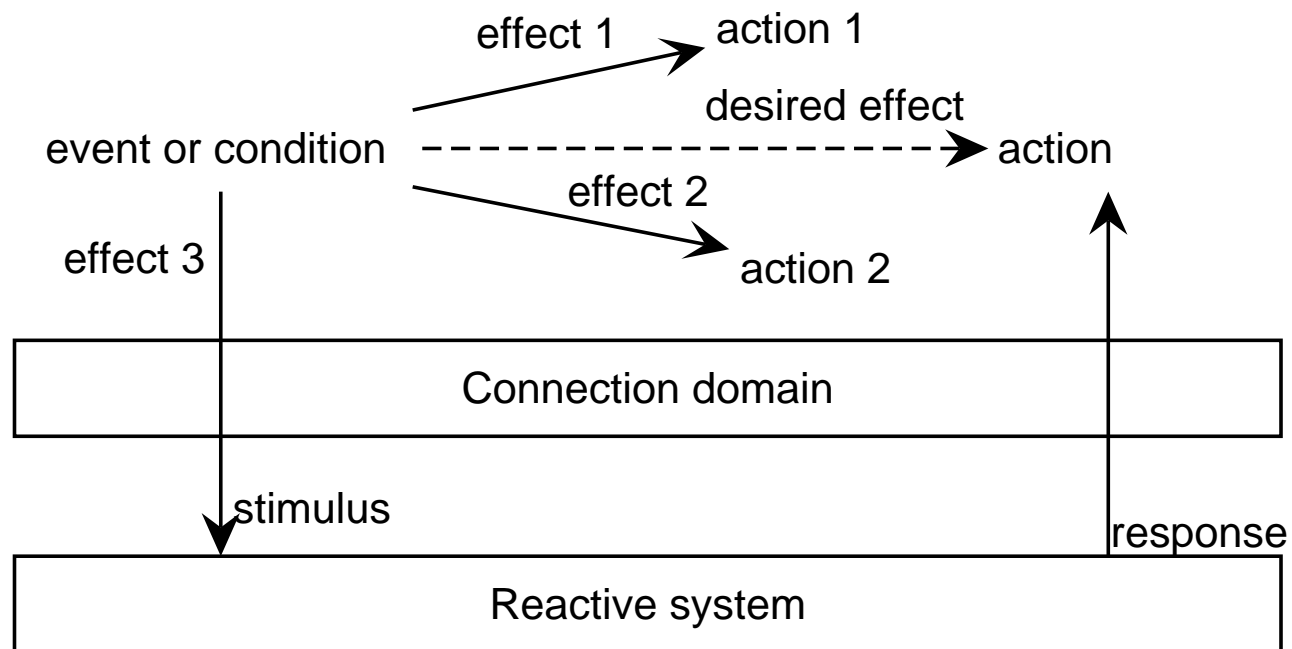


- **Systems engineering argument:** (System specification) and (Assumptions about environment) entail (Emergent properties of composite system).
- To find SuD specification, (1) describe desired emergent behavior, (2) derive system specification, (3) accumulate necessary assumptions when system behavior is not sufficient to produce desired emergent behavior.

Road map



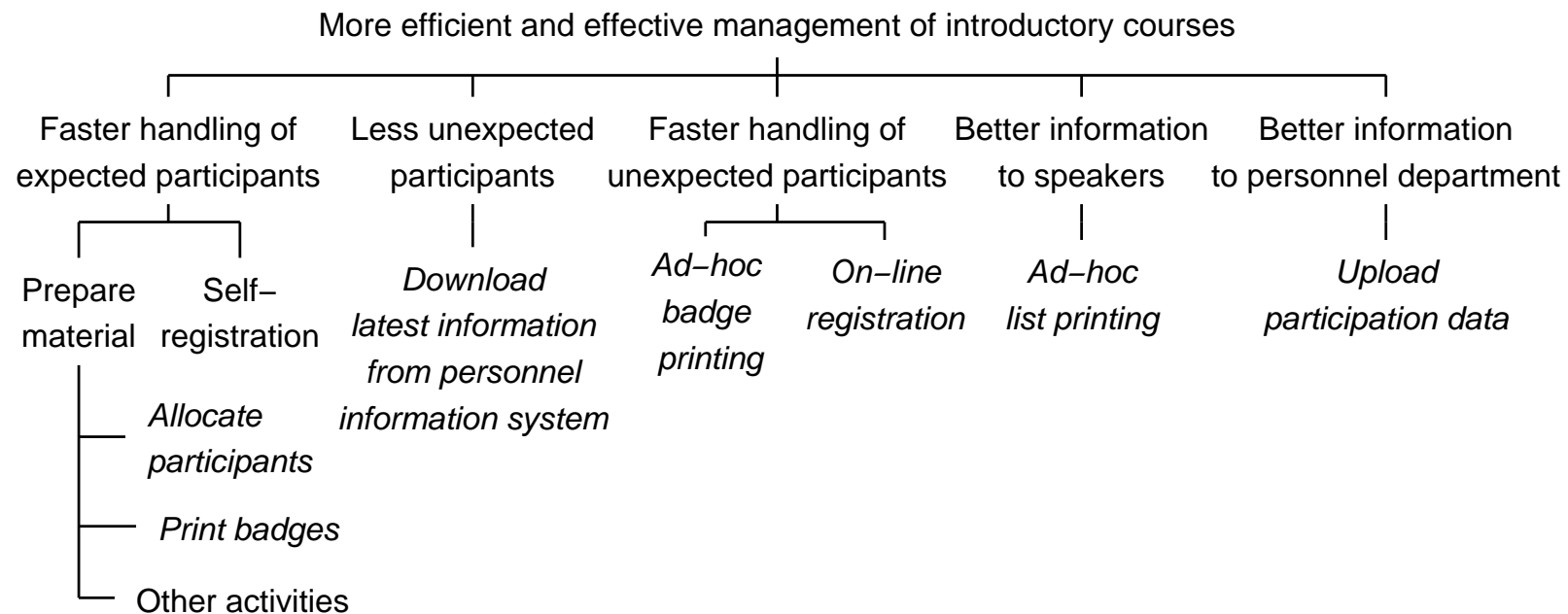
Bringing about desired emergent behavior



- The role of a reactive system is to bring about desired effects in the environment.
- Informative, directive, manipulative functions.

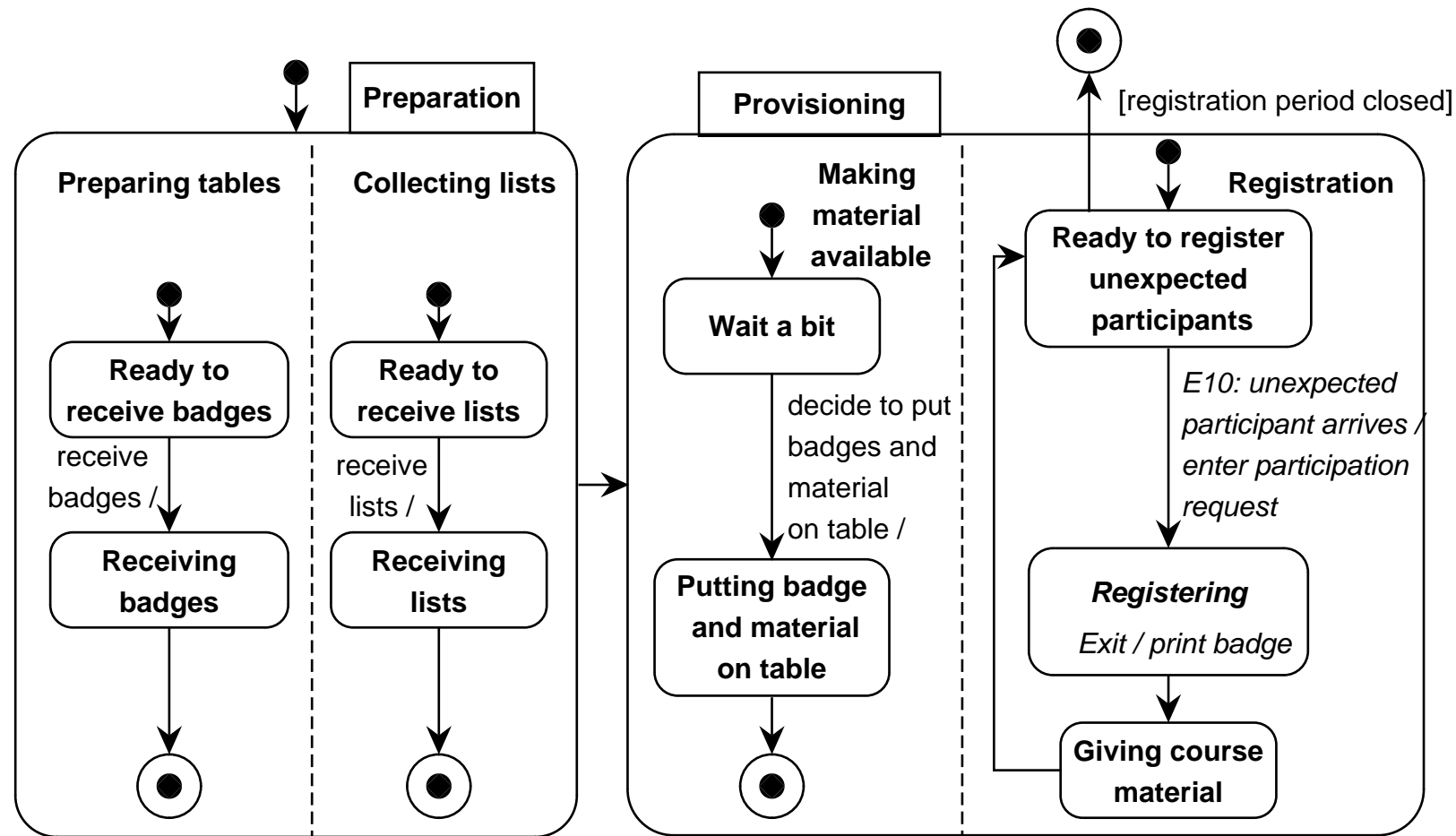
Example 1: Training department

Business goal tree of Training Department



Example 1: Training department

Workflow at the registration desk

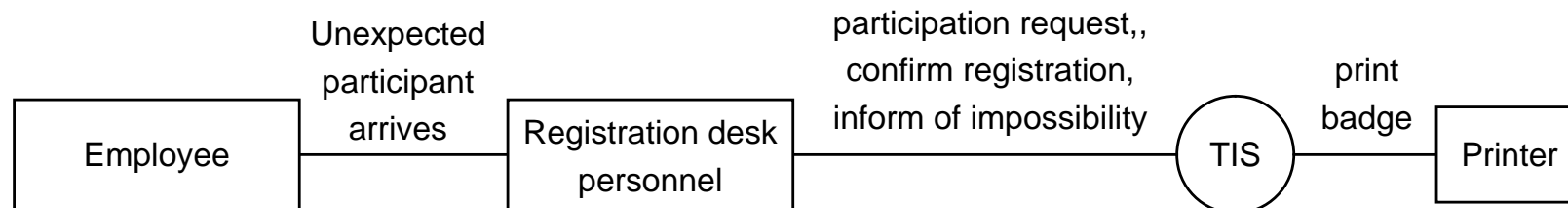


Example 1: Training department

Desired emergent behavior

Event	Desired action
E10 Unexpected participant arrives	If there is still room, the participant should be registered and a badge should be printed.

Embedding the system in its environment



Example 1: Training department

Desired stimulus-response behavior of system

Stimulus	Current state	Response	Next state
S10 Request to register participant is entered.	According to the data of the system, there is still room.	Allocate participant, inform to registration desk personnel, and send badge printing command to printer	System contains updated list of participants.
	According to the data of the system, there is no room.	Inform to registration desk personnel of impossibility.	System contains same list of participants.

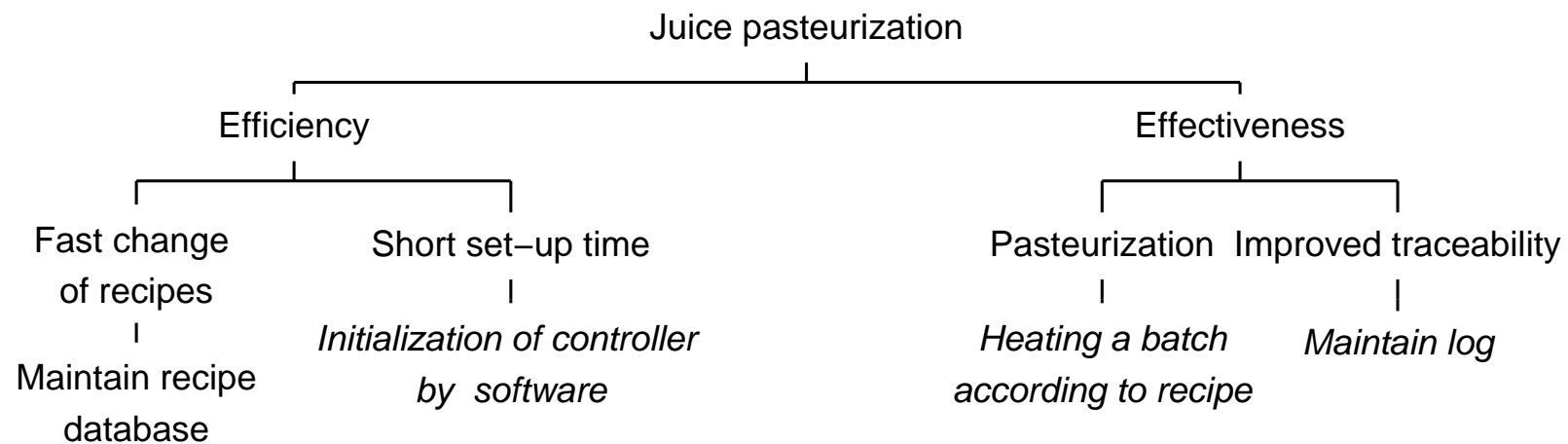
Example 1: Training department

Question

- Which assumptions did we make about the environment?

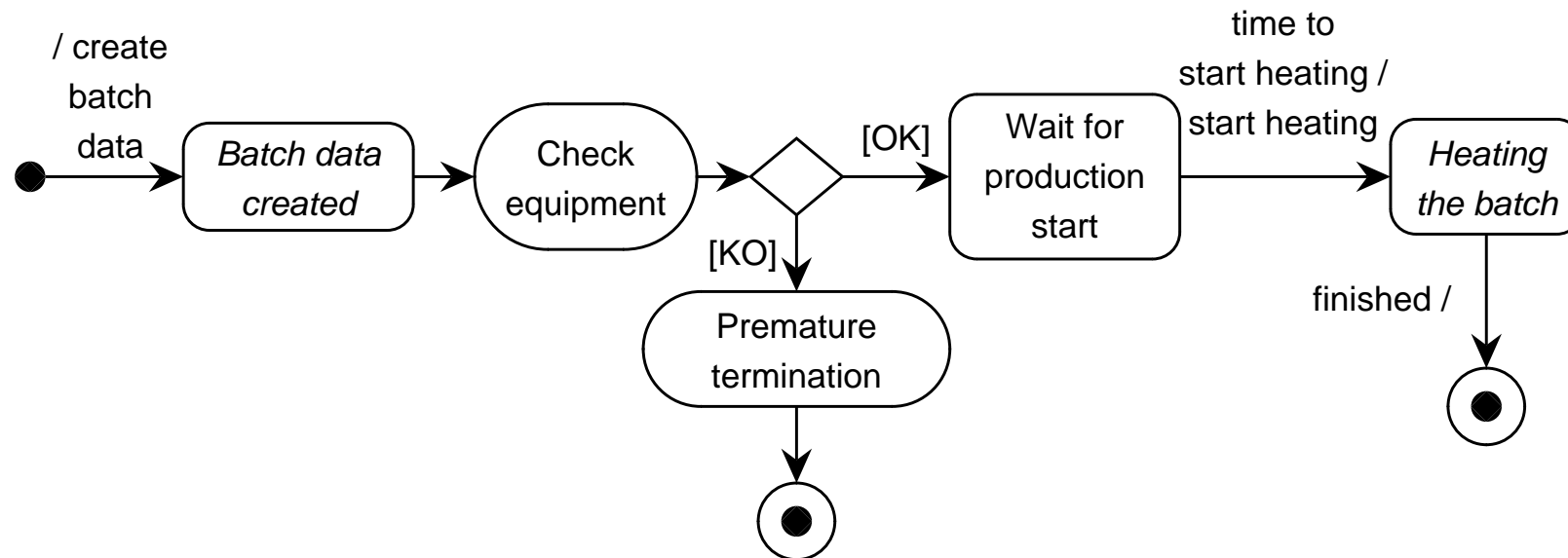
Example 2: Juice pasteurization

Business goal tree of pasteurization department



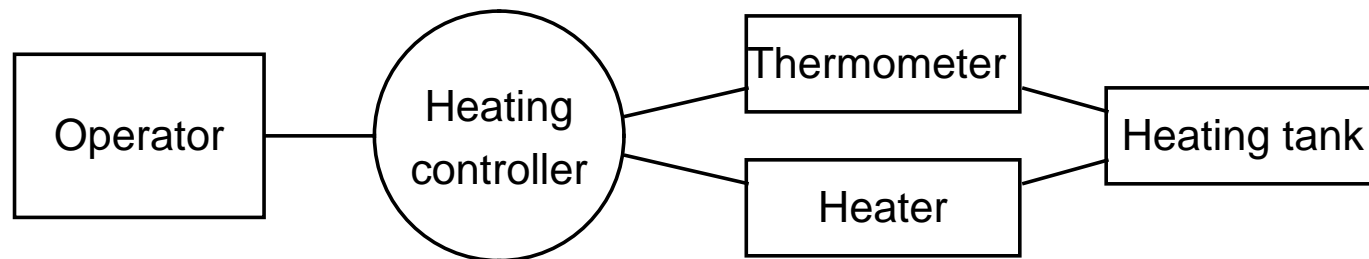
Example 2: Juice pasteurization

Operator workflow



Example 2: Juice pasteurization

Embedding the controller in its context



Example 2: Juice pasteurization

Desired emergent behavior

Event	Desired action
Operator gives command to start heating batch b.	A heating process for the heating tanks of b is started. If at the start of the process, temperature in a tank is too low, the heater of that tank is switched on. When during the process, a tank becomes 5 degrees Celsius warmer than the desired temperature, its heater must be switched off. When it becomes 5 degrees Celsius colder than the desired temperature, its heater must be switched on. When the heating process has lasted for the duration of the recipe, heating must stop and the operator must be notified of this fact.

Example 2: Juice pasteurization

Desired emergent behavior: Transactions

Event	Subject domain state	Desired action
E1 Operator gives command to start heating batch b		Heaters of tanks of b that are below recipe temperature, are switched on.
E2 Temperature in tank t rises 5 degrees above recipe temperature	The juice in t is being heated.	The heater of t is switched off.
E3 Temperature in tank t falls 5 degrees below recipe temperature	The juice in t is being heated.	The heater of t is switched on.
E4 The heating duration has passed, counted since the start of heating of b.	b is being heated.	<ul style="list-style-type: none">• Heaters of b that are on, are switched off.• Operator is informed.

Example 2: Juice pasteurization

Required controller behavior

Stimulus	Current controller state	Desired response	Next state
S1 Operator gives command to start heating batch b	Not heating t and not heating b.	Switch on heaters of tanks with low temperature.	Heating t and b.
S2 Every 60 seconds.	Heating t and measured temp $>$ desired temp + 5	Controller switches off the heater of t.	Heating t.
	Heating t and measured temp $<$ desired temp - 5	Controller switches on heater of t.	Heating t.
S4 Recipe time since the start of heating of b has passed.	Heating b.	<ul style="list-style-type: none">• Switch off heaters of b that are on.• Inform operator.	Not heating b.

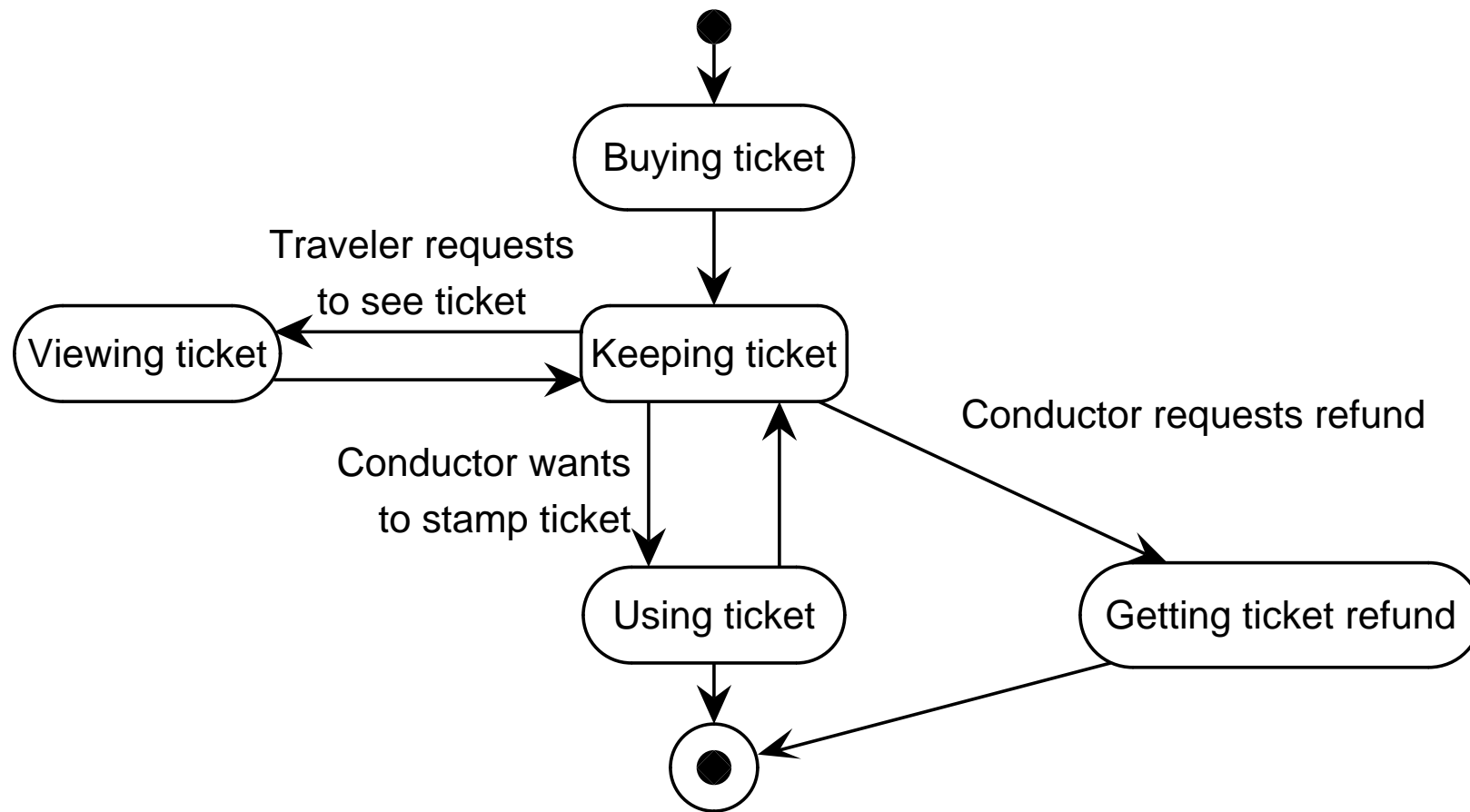
Example 2: Juice pasteurization

Assumptions about environment

- The operator works according to the operator workflow.
- The operator only gives the start command when the batch is in its heating tanks.
- The heater of a tank never breaks.
- The controller is connected to the heaters of the tanks.
- The thermometer of a tank never breaks.
- The controller is connected to the thermometers of the tanks.

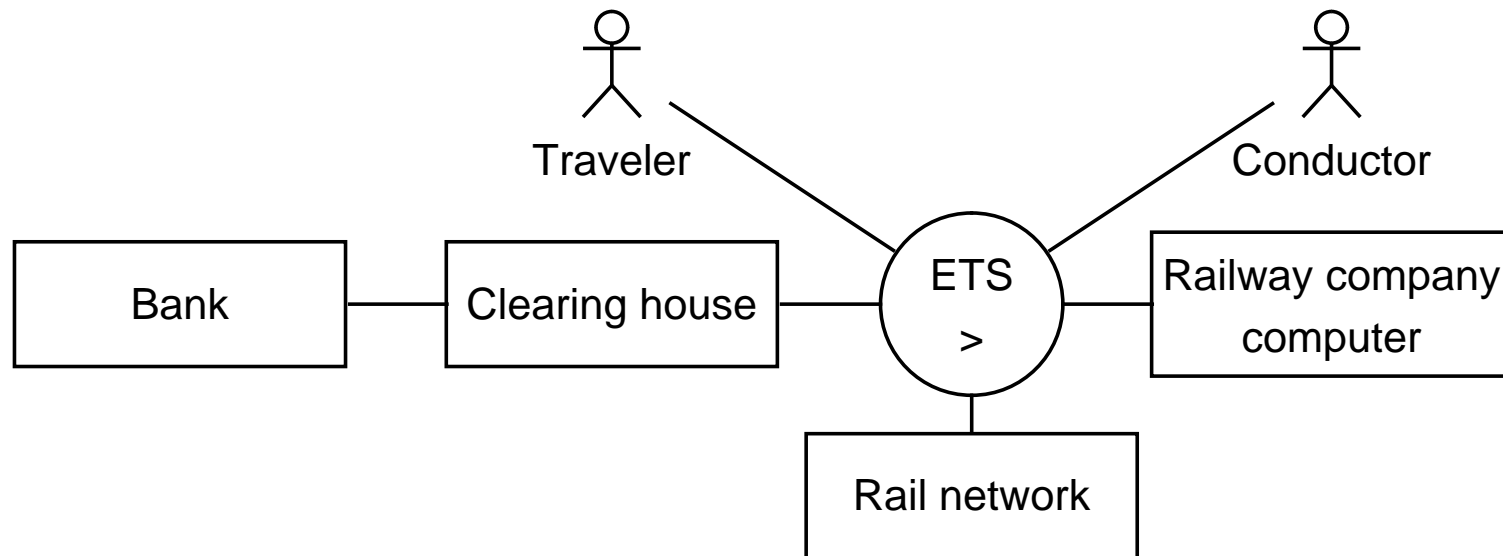
Example 3: Electronic Ticket System

“Workflow” of a ticket



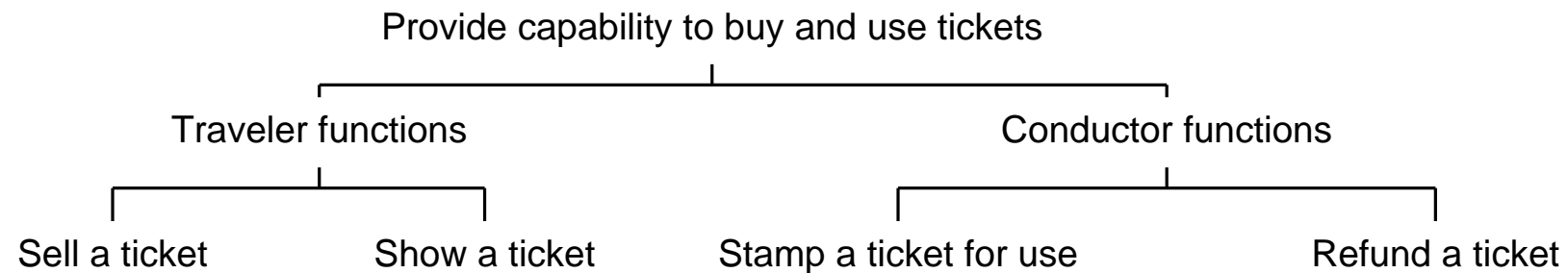
Example 3: Electronic Ticket System

The ETS in its logical context



Example 3: Electronic Ticket System

Required ETS functions



Example 3: Electronic Ticket System

Required ETS services

- | |
|--|
| <ul style="list-style-type: none">• Name: Sell a ticket.• Triggering event: Traveler requests to buy a ticket.• Delivered service: Allow a traveler to buy a ticket. |
| <ul style="list-style-type: none">• Name: Show a ticket.• Triggering event: Traveler requests to view a ticket.• Delivered service: Display ticket attributes to the user. |
| <ul style="list-style-type: none">• Name: Stamp a ticket for use.• Triggering event: Request to stamp an unused ticket part.• Delivered service: Mark the requested part of the ticket as used. |
| <ul style="list-style-type: none">• Name: Refund a ticket.• Triggering event: Request to refund an unused ticket part.• Delivered service: Cause a refund and make invalid. |

Example 3: Electronic Ticket System

Other desired properties

- A traveler cannot get a ticket without paying for it.
- The price of a ticket is withdrawn from the bank account associated with the smart card.
- A traveler who has paid for a ticket gets it.
- A refunded ticket cannot be used any more.
- A fully used ticket cannot be refunded.
- It is not possible to use a ticket twice.

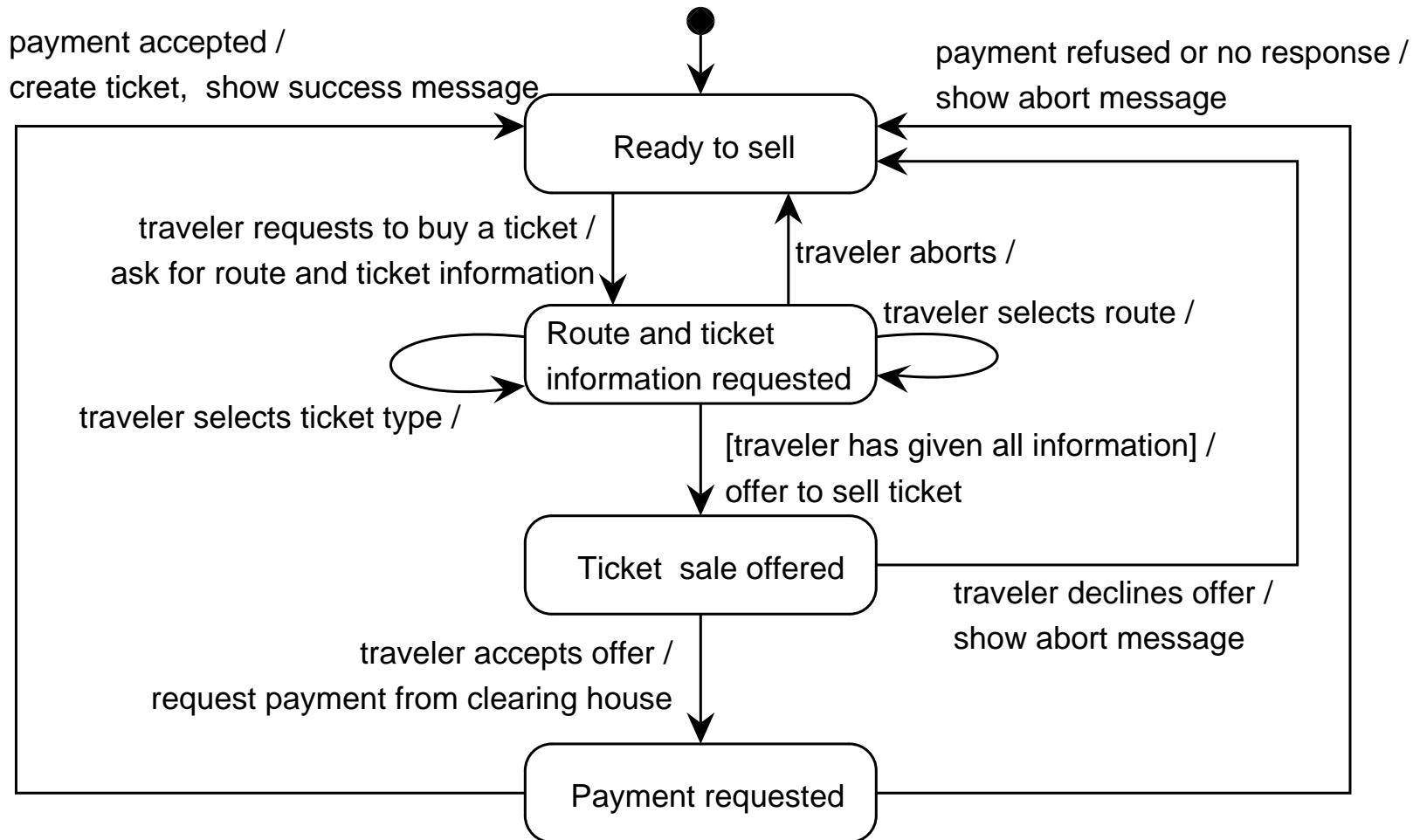
Example 3: Electronic Ticket System

Desired emergent behavior

Event	Current domain state	Desired action	Next domain state
Request to buy ticket	Any	Ticket created and paid for	Ticket and payment exist
Request to see tickets	Any	All tickets are displayed	Unchanged
Request to use ticket for a rail segment	Ticket exists and is valid for that rail segment	Create ticket stamp for that rail segment	Ticket exists and ticket usage stamp exists for this rail segment and ticket
Request to refund ticket	Ticket exists and not completely used	Create refund stamp	Ticket refunded and ticket is not longer valid

Example 3: Electronic Ticket System

Selling scenario



Example 3: Electronic Ticket System

What is so special about this example?

- Some subject domain entities are virtual.
- They are implemented by the system
- So some required system behavior *is* desired subject domain behavior (rather than causing it indirectly).

Summary of guidelines (1)

Finding states:

- ✓ Look for states in which the behavior is waiting for something to occur.
- ✓ Look for modes of behavior (e.g. normal-standby-emergency etc.)

Finding events:

- ✓ Look for signals, condition changes, temporal events to which the system must respond.
- ✓ Look for desired effects of the system. What triggers the system to produce such an effect?

Summary of guidelines (2)

Dealing with parallelism:

- ✓ If you cannot show with absolute certainty that two events occur in sequence, then assume they occur in parallel.

Introducing hierarchy:

- ✓ Introduce hierarchy to express commonality in behavior.
- ✓ Introduce hierarchy to introduce common response to some event (e.g. alarm, suspend, mode change, etc.)

Summary of guidelines (3)

Finding system behavior:

- ✓ Follow several paths through our road map and check whether they lead to the same behavior specification.
- ✓ Through desired causality to system transactions.
 - Identify business solution goals.
 - Identify solution activities in subject domain, user workflow.
 - Identify desired event-action in the environment.
 - Map to stimulus-response pairs of SuD; introduce connection domain if necessary.
 - Refine to transactions by introducing SuD states.

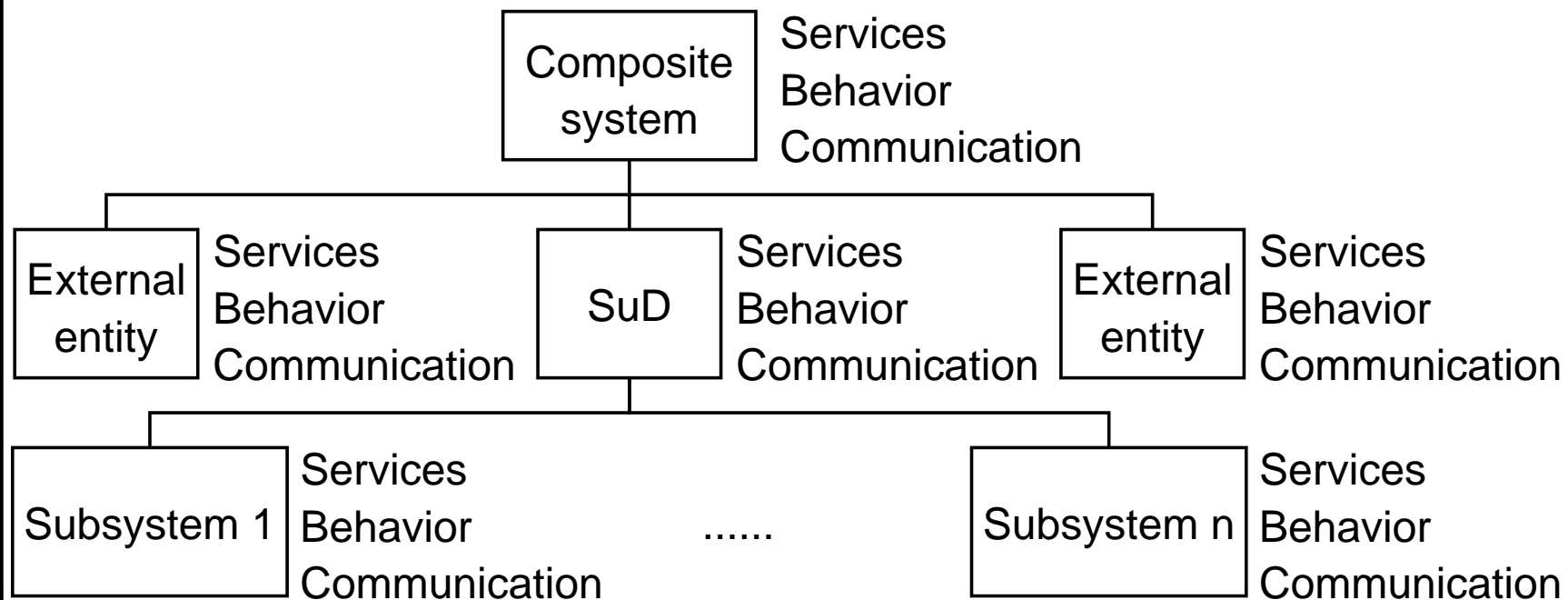
Summary of guidelines (4)

- ✓ Through desired system services to system transactions.
 - Identify business solution goals.
 - Derive desired system services: triggers and added value, assumptions.
 - Include connection domain.
 - Refine to stimulus-response pairs; Introduce system states

Behavior Modeling and Design Guidelines: Main Points

- To find required system behavior, look for desired emergent composite system behavior.
- Reduce this to desired system behavior.
- Accumulate any assumptions about environment needed to show that desired system behavior is sufficient to create desired emergent behavior.

Part V. Communication Notations



Uses of communication notations

Context modeling

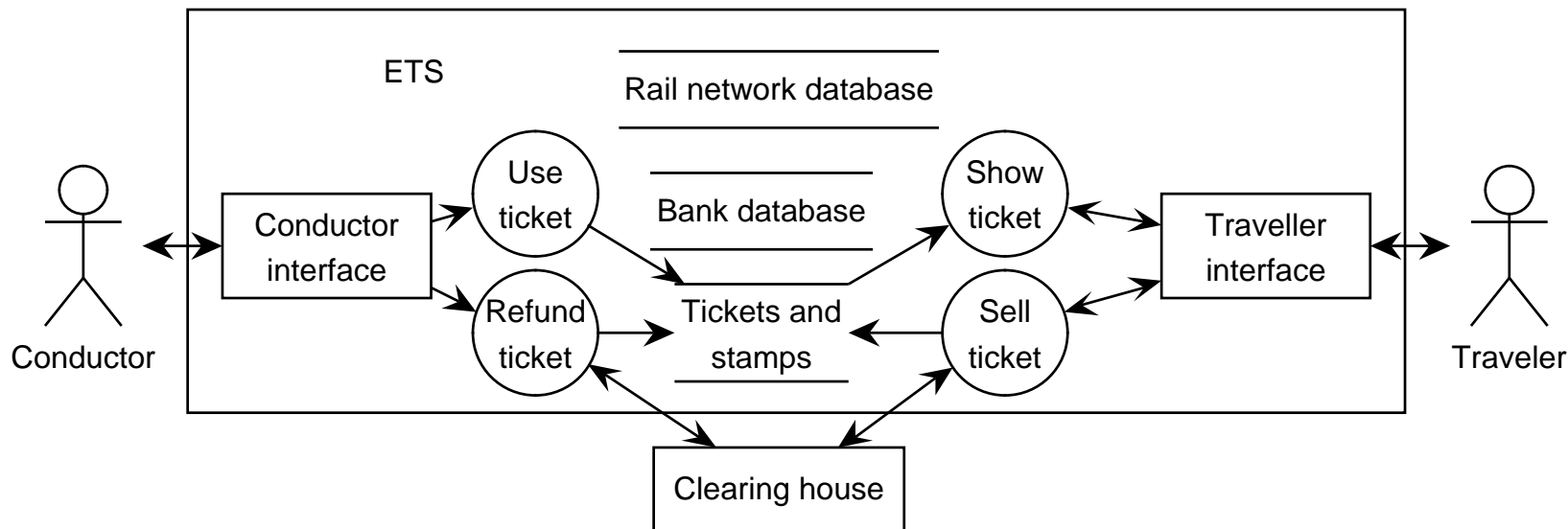
- Communications in environment
- Communications between SuD and environment

Architecture design

- Decomposition of stimulus-response pairs into communication between components
- Communications between components and external entities

Example: Electronic Ticket System (1)

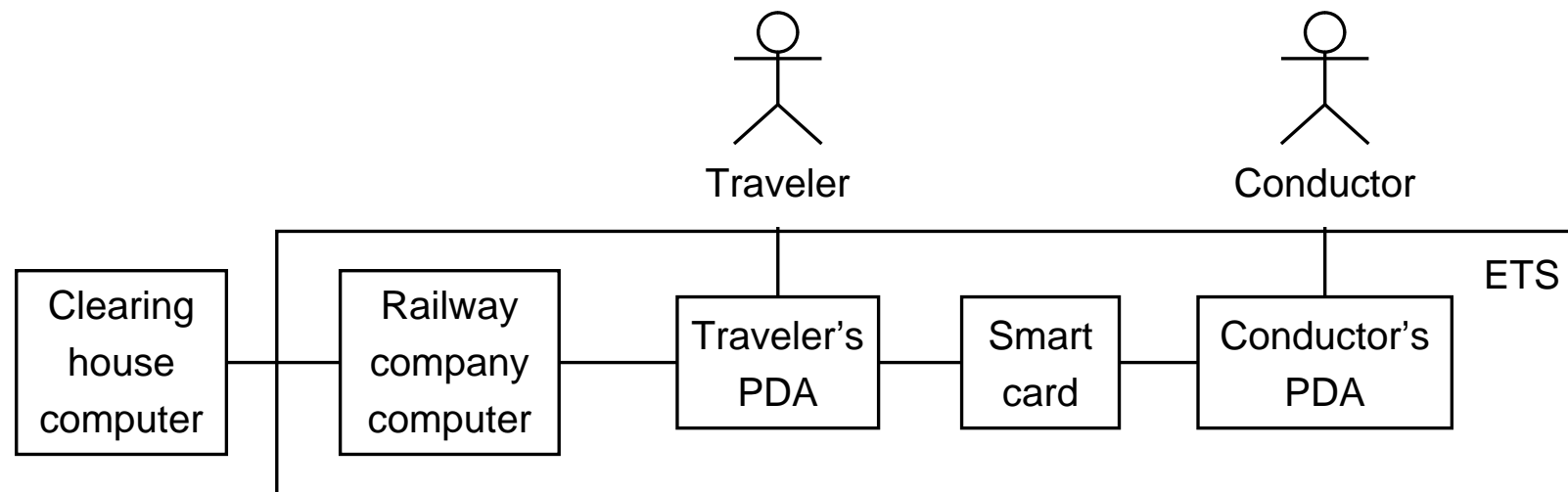
Requirements-level architecture



- Independent from physical architecture
- Independent of software implementation platform
- Motivated only in terms of environment and requirements

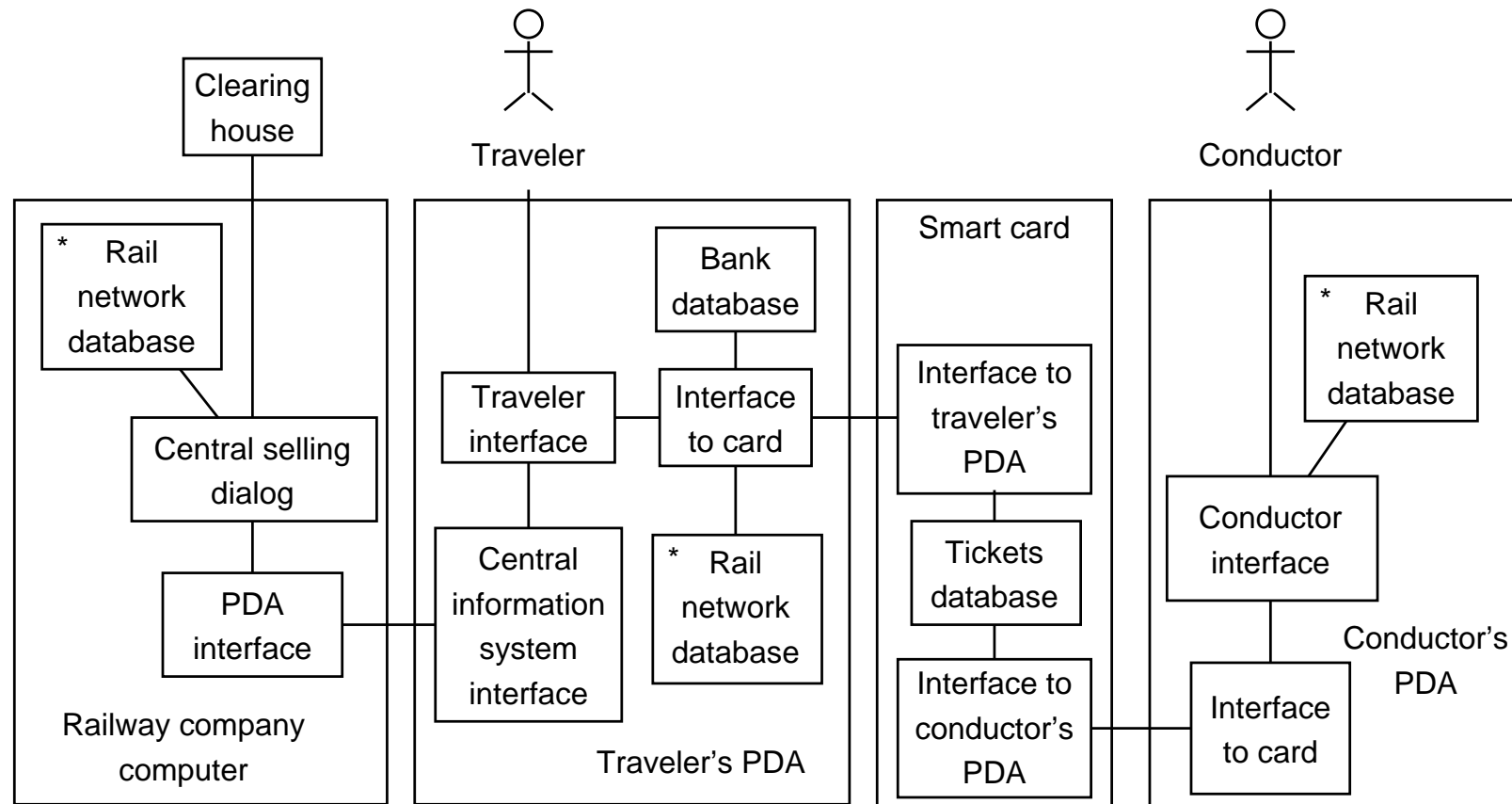
Example: Electronic Ticket System (2)

Physical network architecture



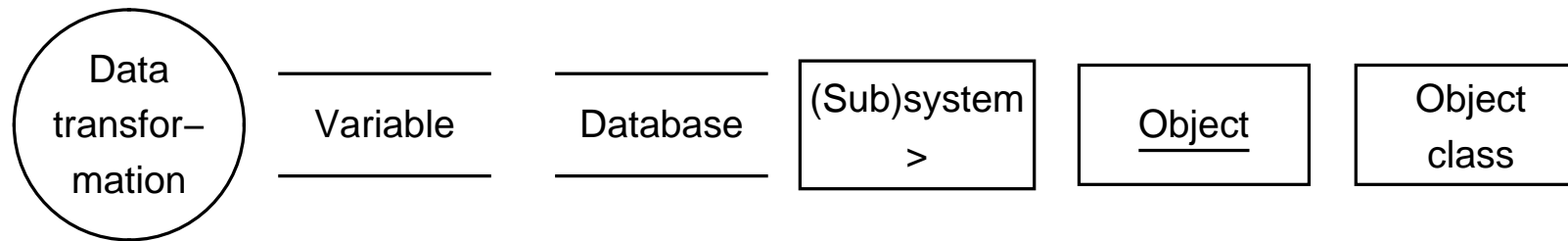
Example: Electronic Ticket System (3)

Allocation of essential components to physical components



Components introduced to deal with internal interfaces. Replicated components indicated by asterisk.

Icons used



- There is a small number of icons used to represent the communication between components in an architecture.
- Data flow diagrams (DFDs): No boxes.
- Architecture diagrams: All icons.

Structure of part IV

- Data flow diagrams (DFDs)
- Architecture diagrams
- Execution semantics
- Context modeling guidelines
- Architecture design guidelines

Main points

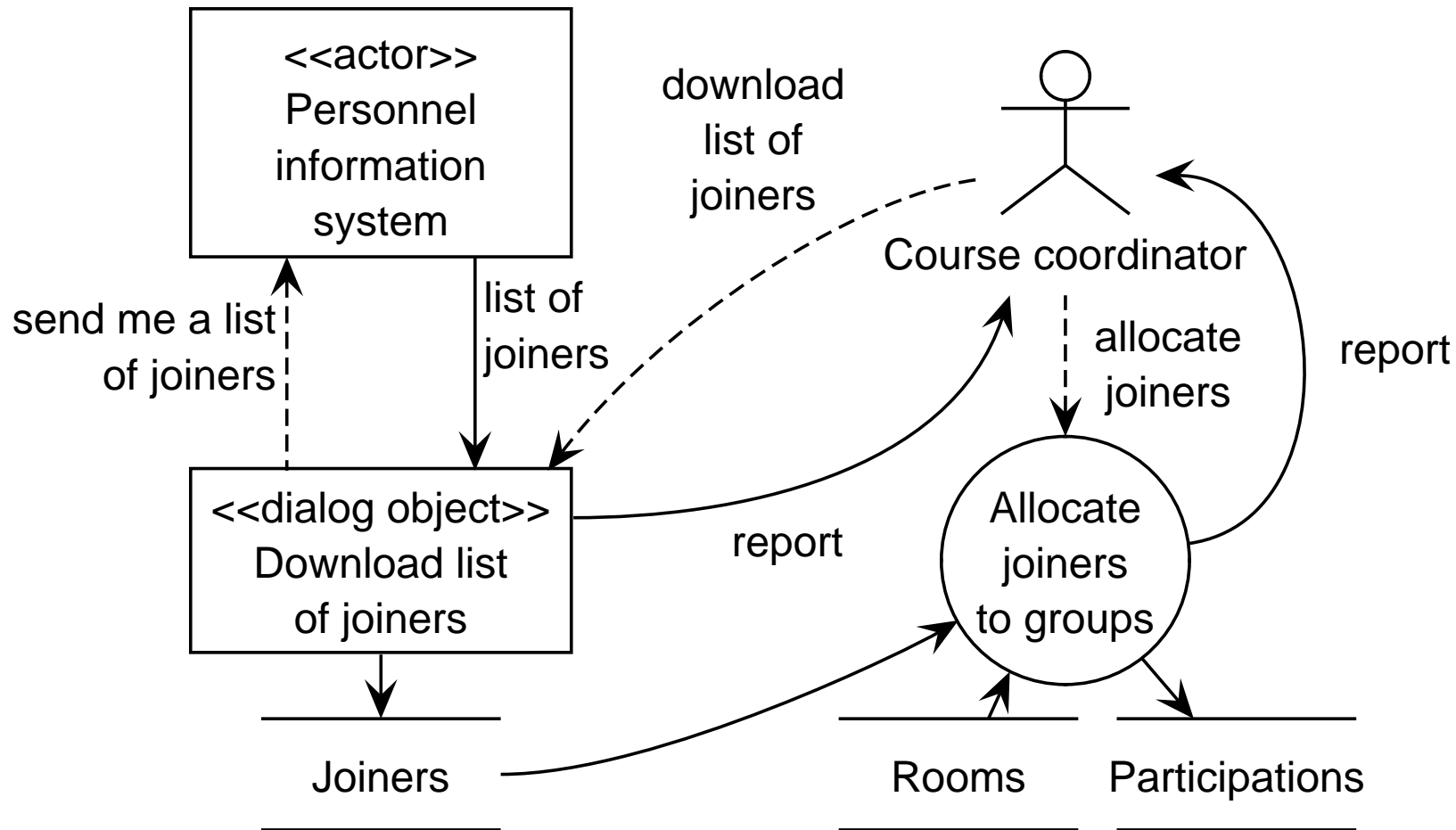
- Communication occurs in the environment and at all levels in the system.
- Find essential system decomposition by
 - Analyzing required external system communication and
 - modeling desired environment communications.
- Communications can be described by DFDs or architecture diagrams.

Chapter 15. Data Flow Diagrams

- Introduced in the 1970s to represent logical software decomposition.
- Still widely used.
- Ontology built into the notation: Software consists of data stores and data transformations.

(Ontology = classification of kinds of things.)

Fragment of architecture of Training Information System

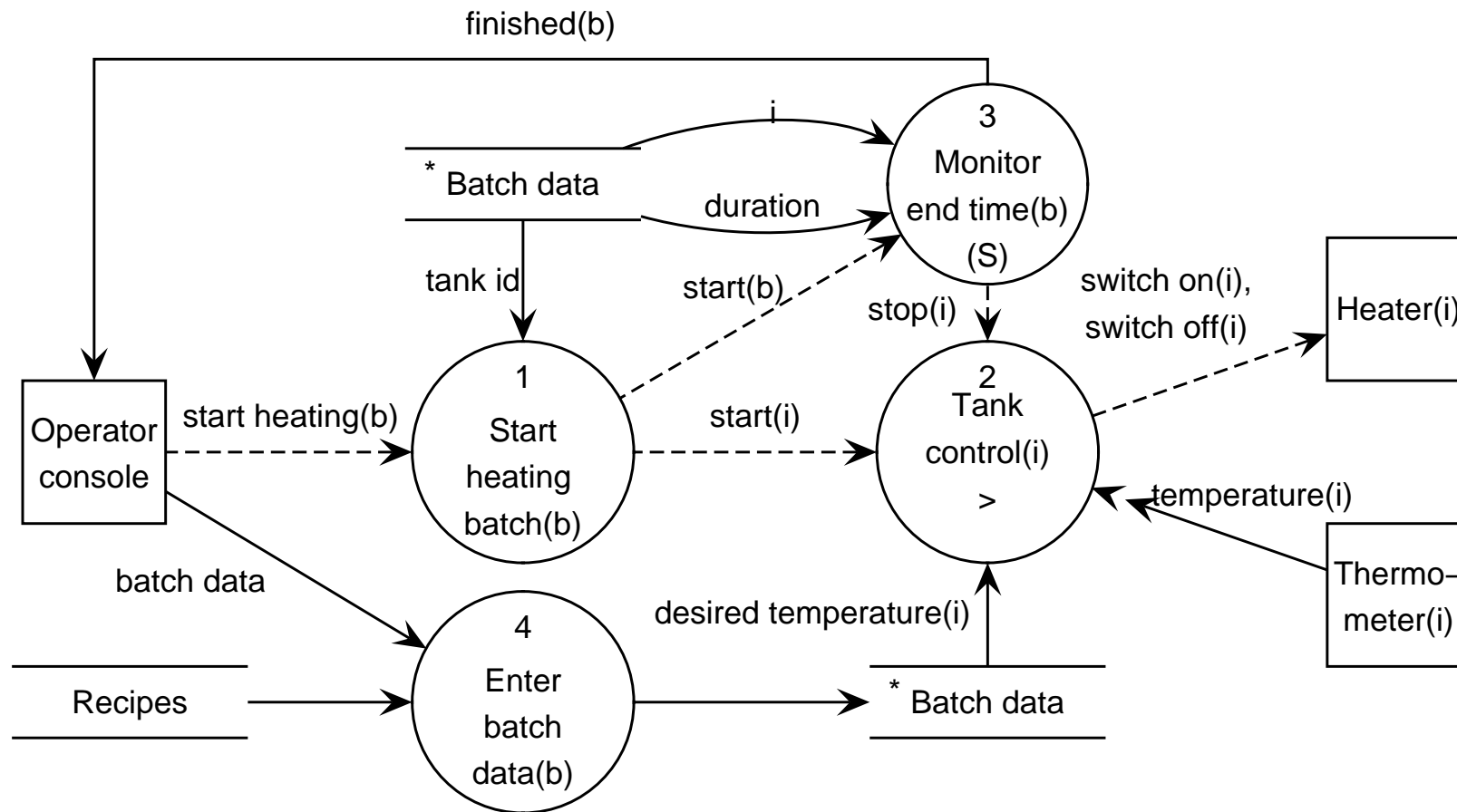


Typical information system architecture

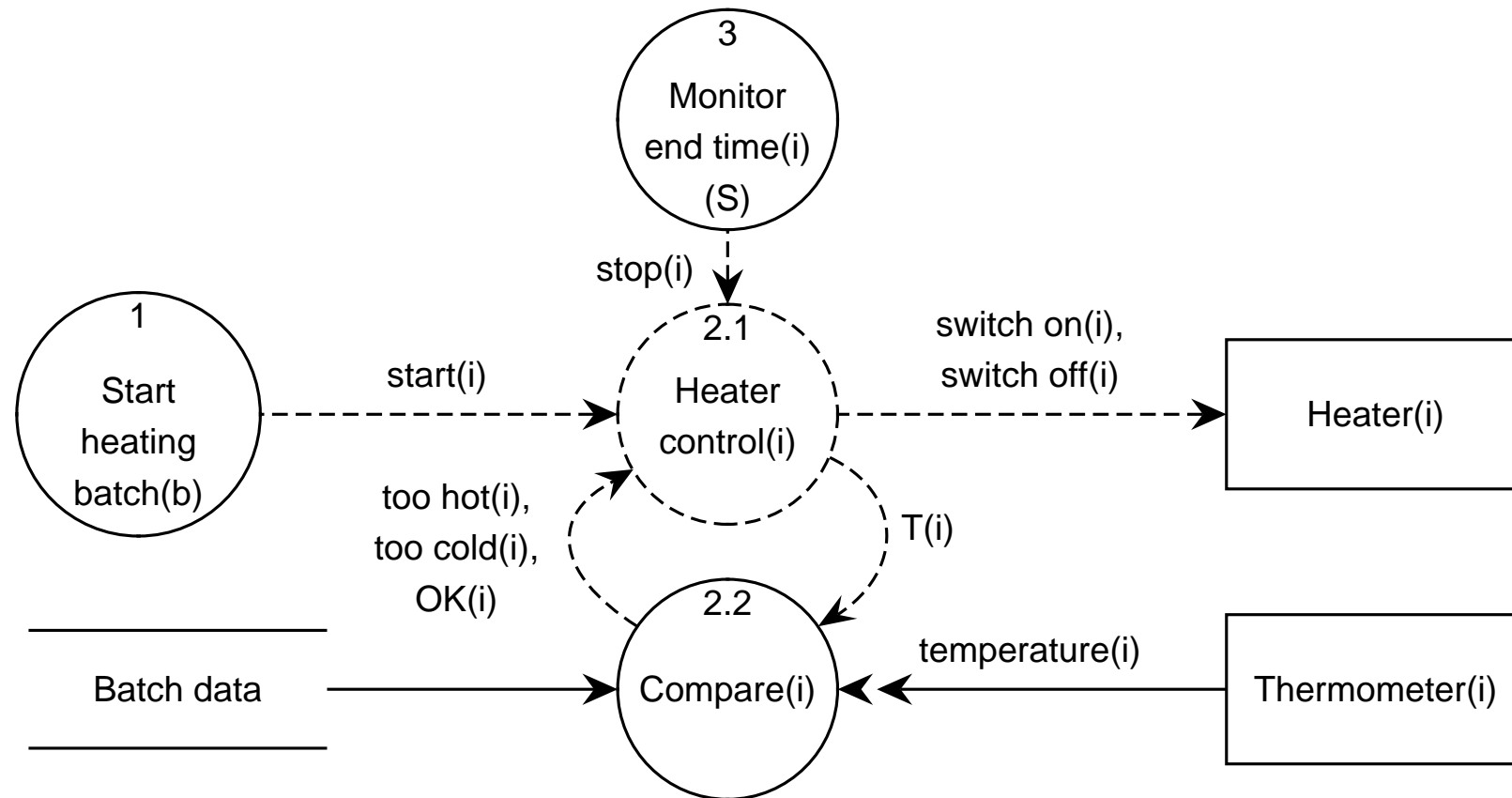
- Collection of data stores holding info about subject domain.
- Collection of data transformations accessing the stores on behalf of external entities.

Often more insightful to present functional DFD fragments separately.

Requirements-level architecture of heating controller



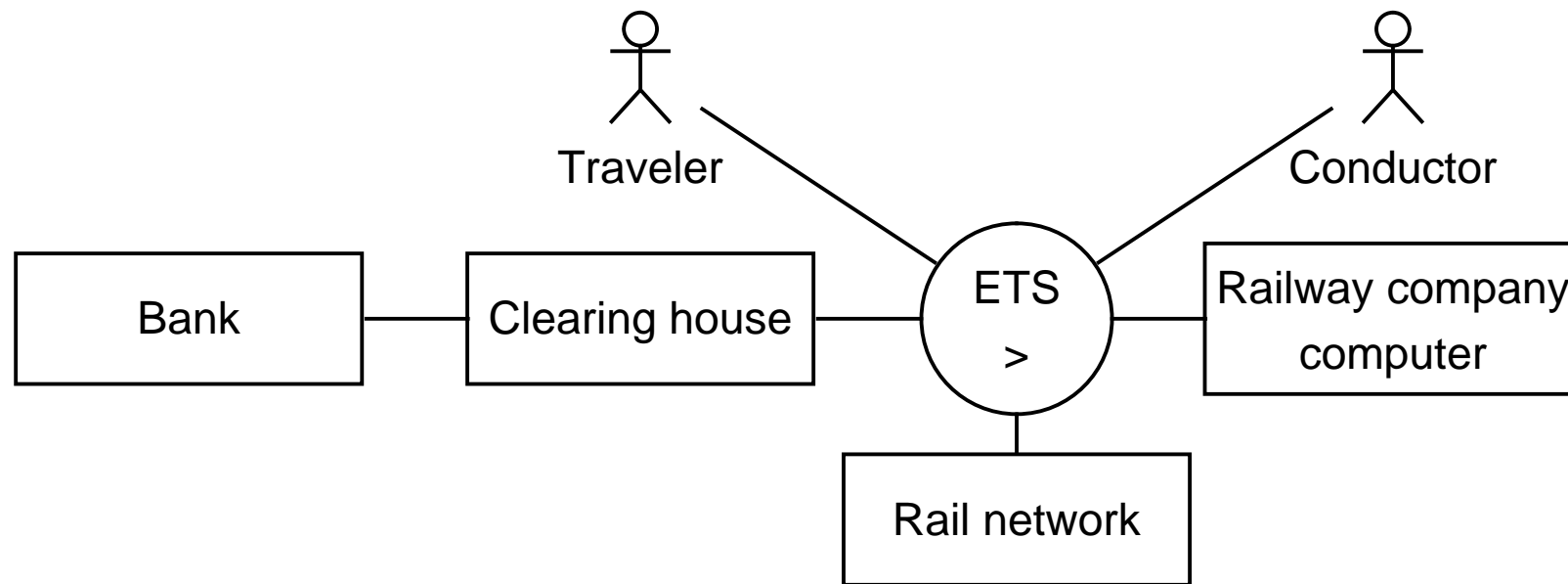
Decomposition of Tank control(i)



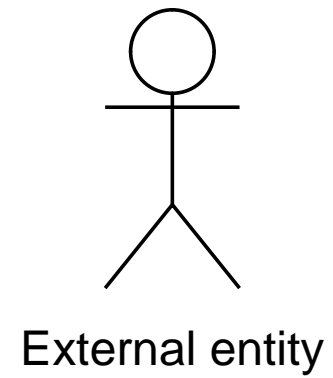
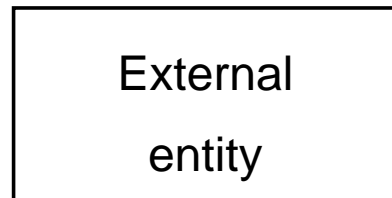
Typical control system architecture

- One or more control processes.
- Collection of interface processes connecting the control processes to external devices.

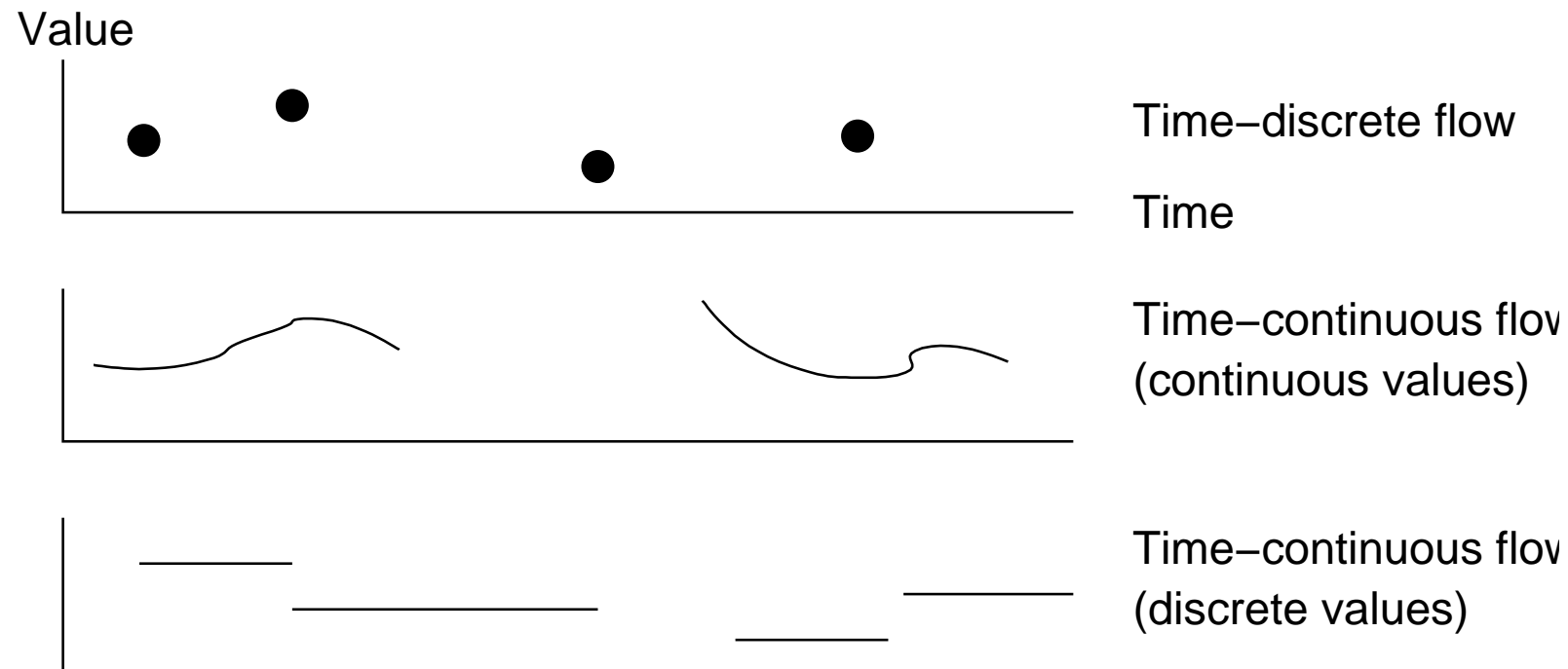
Context diagram of ETS



Icons for external entities



Time-behavior of flows



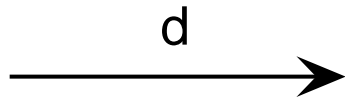
NB. Time-discrete and time-continuous flows all carry data values.

Icons for flows



Communication channel

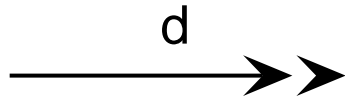
Connected entities can communicate.



Time-discrete data flow

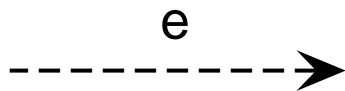
Sender can share info with receiver.

Data is present at discrete instants of time.



Time-continuous data flow

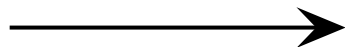
Data present during periods of time



Event flow

Sender can cause receiver to do something.

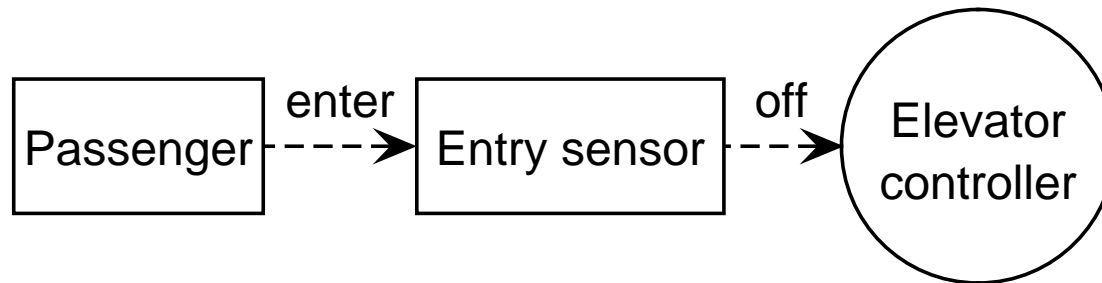
Named after effect or after cause.



Compound flow

Bundles a collection of time-discrete flows.

Event flows

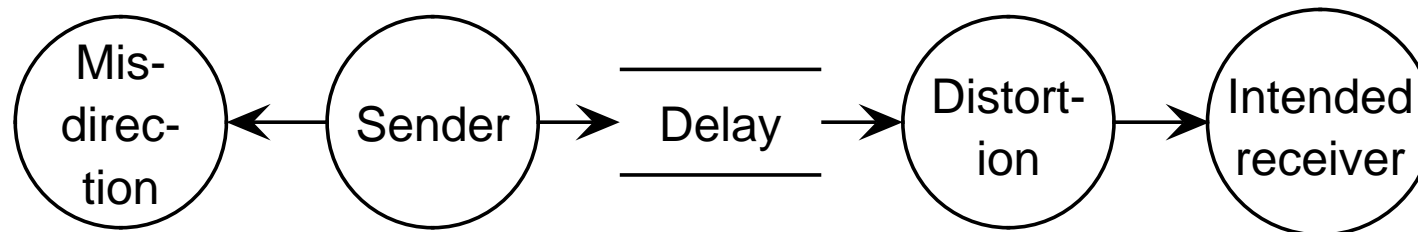


- In our approach, event flows can carry events with parameters.
- Not allowed in Yourdon approaches.
- In our approach, the difference between event flows and data flows is only in the naming.

Communication semantics of flow lines

A **flow** is an instantaneous and reliable communication channel between two elements.

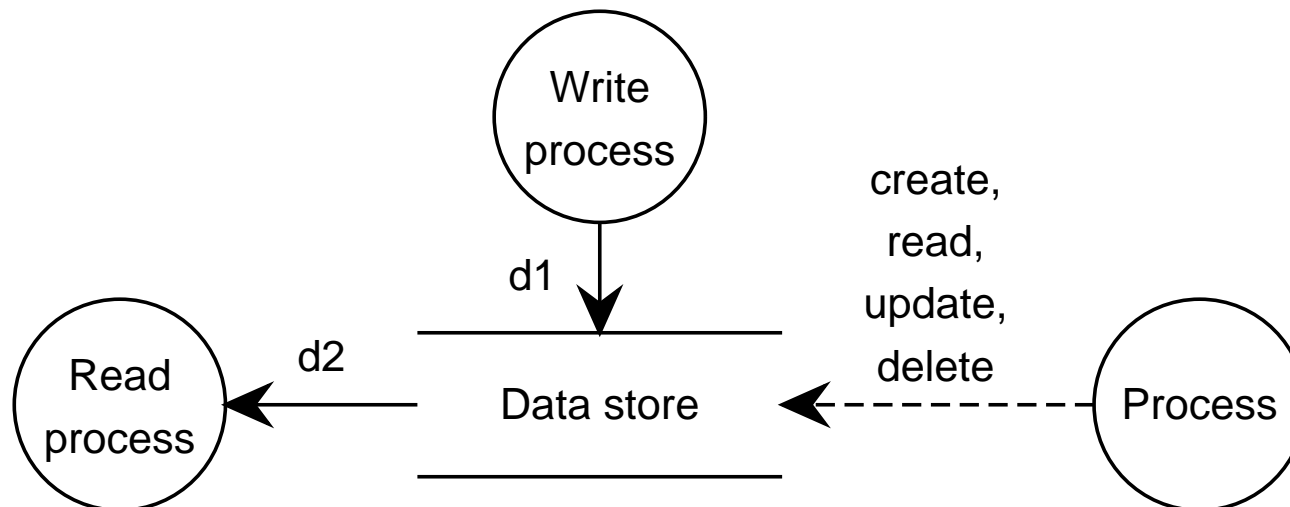
- Introducing delay and unreliability:



- A flow is an abstraction!
- We always must choose *some* abstraction level.

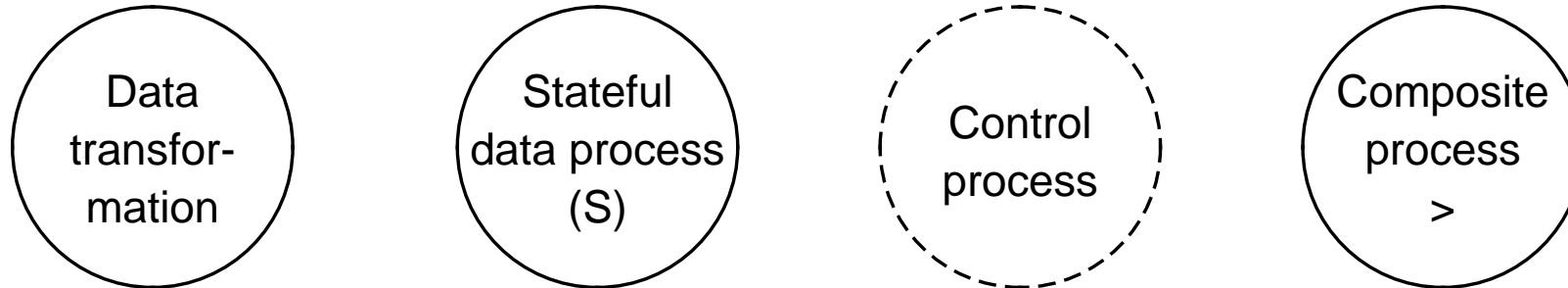
Stores

Remembers data until deleted explicitly.



- NB. Event flow access not allowed in Yourdon methods.
- Conceptual structure of subject domain data is represented by subject domain ERD.

Processes



- **Data process** transforms input data into output data.
 - *Stateless:* Data transformation. Can be triggered by a prompt T.
 - *Stateful:* Can be triggered by an enable/disable prompt E/D.
- **Control process** transforms input events into output events. Specified by an STT or STD.
- **Composite process** is specified by a lower-level DFD.

Specification of a data transformation

2.1: Compare(i).

- When triggered, then let d be the desired batch temperature from the Batch data store:
 - if $\text{temperature}(i) < d - 5$ then too cold(i),
 - if $\text{temperature}(i) > d + 5$ then too hot(i).
- Mathematical input-output relation.
- Interface must match DFD.

Specification of stateful data process

3: Monitor end time.

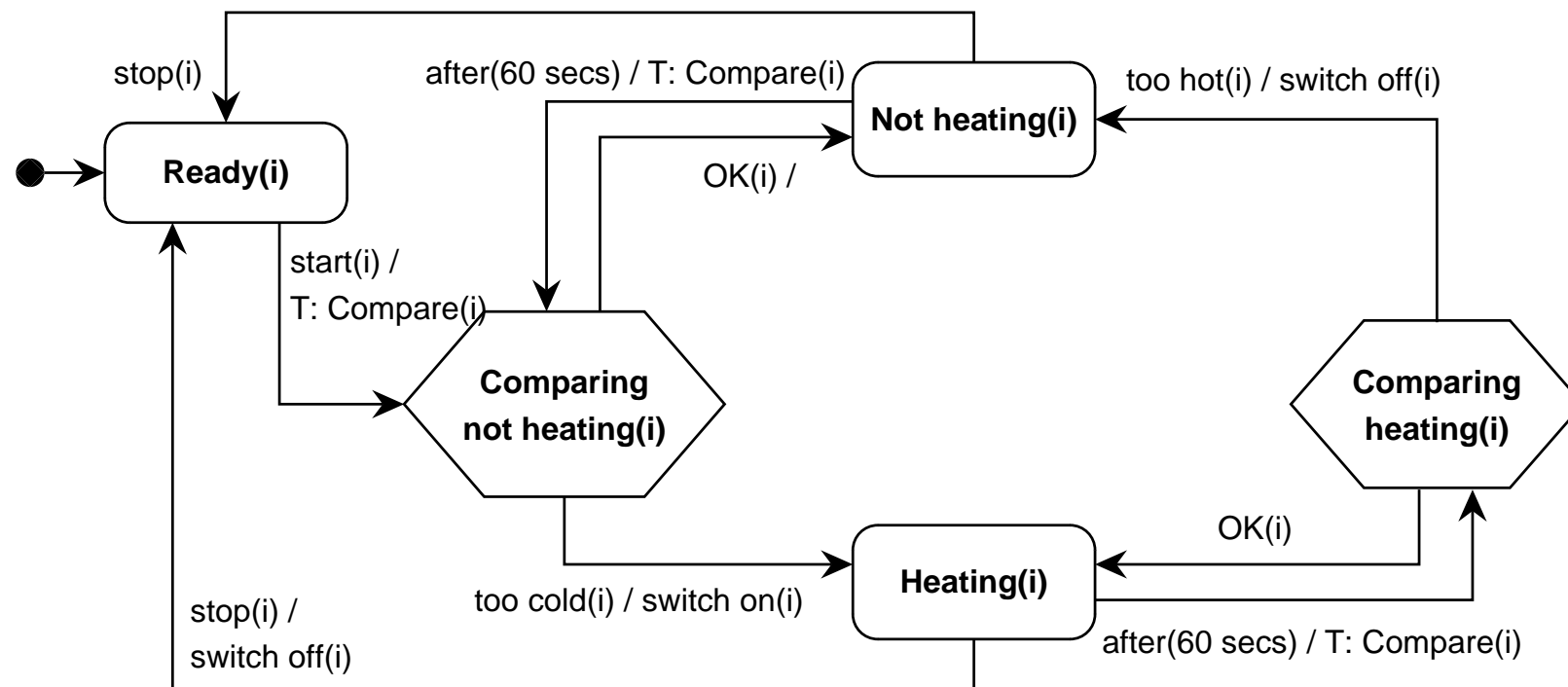
- Local variable: end_time.

This process is started by the reception of b and then is active until a temporal event occurs.

- Initialization: When b is received, then
 - Read duration of b from Batch data and set $\text{end_time} := \text{now} + \text{duration}$.
- Process: When end_time occurs, then
 - read tank_id's of b from batch data,
 - for each tank_id do stop(tank_id),
 - send finished(b).

- Initialization is followed by state-dependent transformation process.
- Interface must match DFD.

Specification of control process by Mealy diagram



- Interface must match DFD.
- Yourdon only allows Mealy diagrams. We allow any STT or STD.

Parametrized DFDs

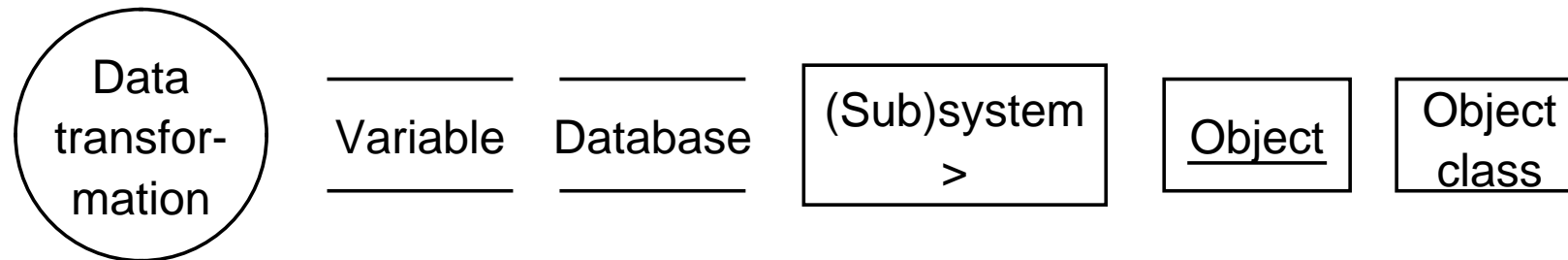
DFD is instance-level diagram. To simulate types of elements:

- Process names in a diagram can be parametrized by a process identifier.
- This causes some flow names to contain process identifiers as well.

Main points

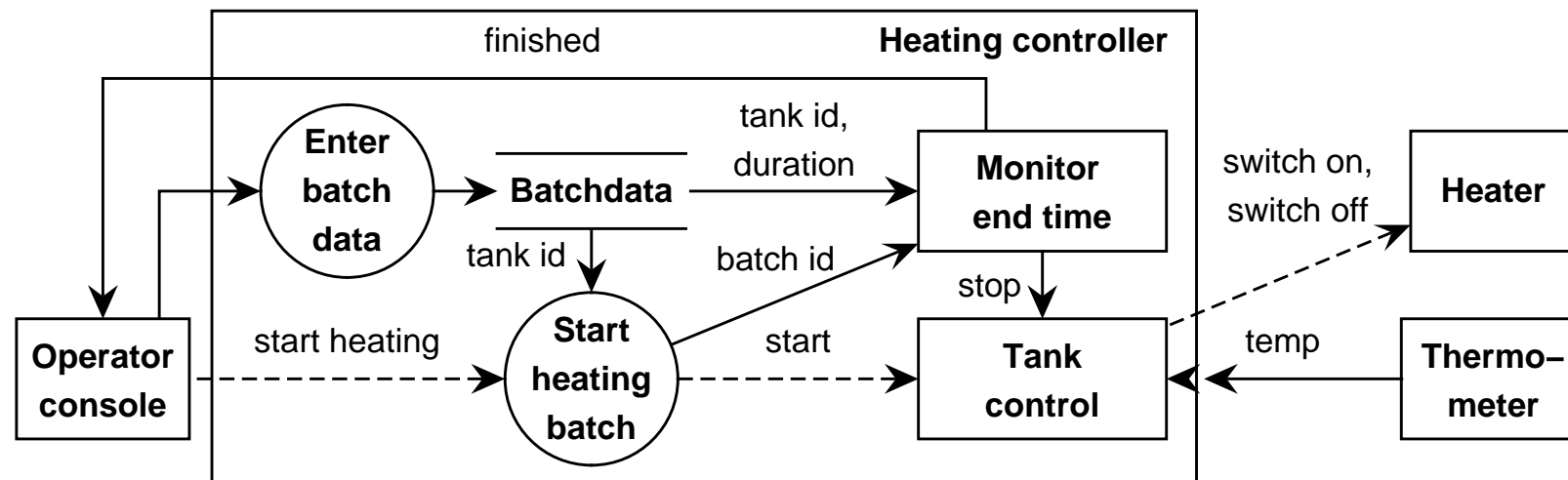
- DFDs model a system as a collection of communicating data stores and processes.
- Various kinds of processes, depending upon how they are specified.
- DFDs can be hierarchical.
- Detailed information about flows can be expressed.
- Instance-level notation.

Chapter 16. Communication Diagrams

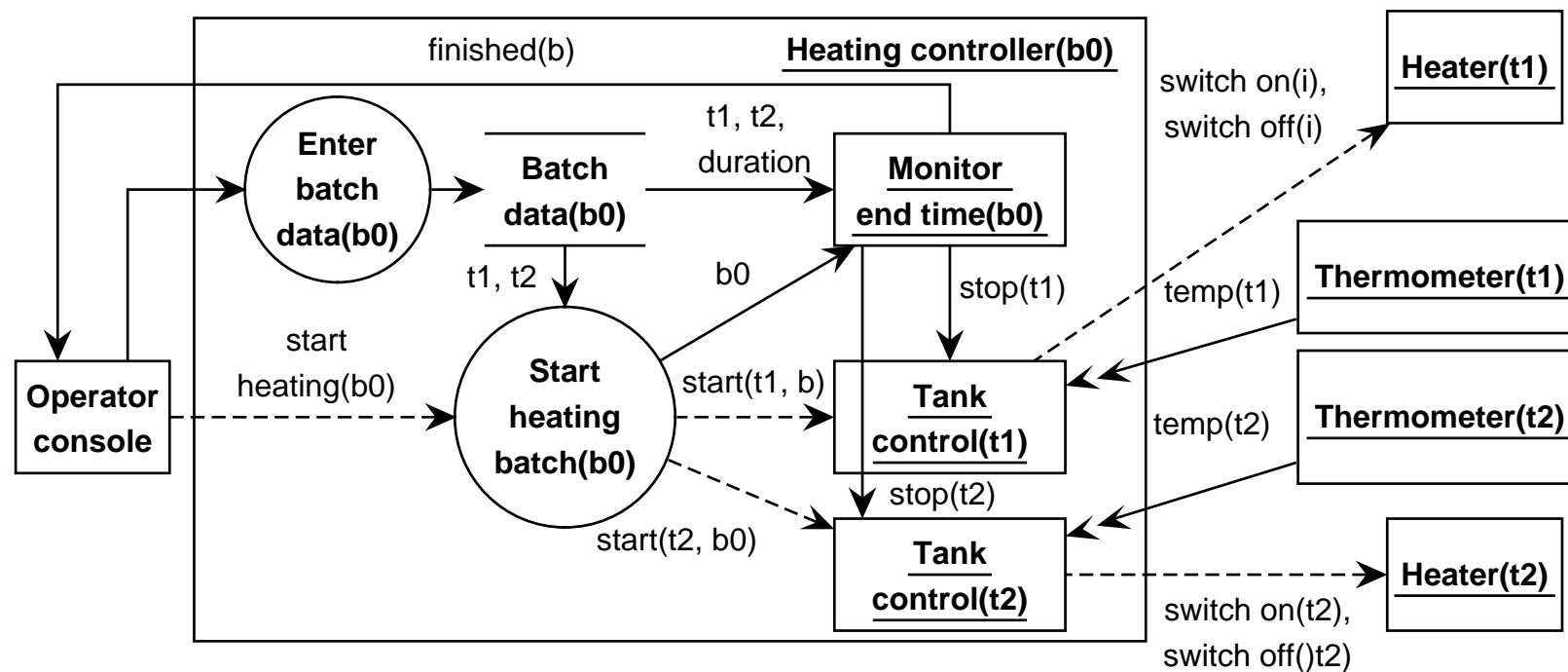


- Difference with DFD: Type-level diagram; more icons.
- Difference between variable and database is that database is (viewed as) a set of instances.
- An object is a subsystem not decomposed by a communication diagram.
- Object classes are not components of a system. They are types of such components.

Communication diagram of heating controller requirements-level architecture



Instance diagram of a particular heating controller in a particular context



Components

Very overloaded term. For us: component = part of an executing SuD that delivers service to its environment.

- Data transformation
- Data store
 - Variable
 - Database
- Subsystem
- Object

To deal with a time-varying collection of components, we can also show object classes. These are not components themselves. (They are *types* of components.)

Communication channels

Same as in DFDs.

- Event channel. Named after cause or effect.
- Data channel. Time-discrete or time-continuous.

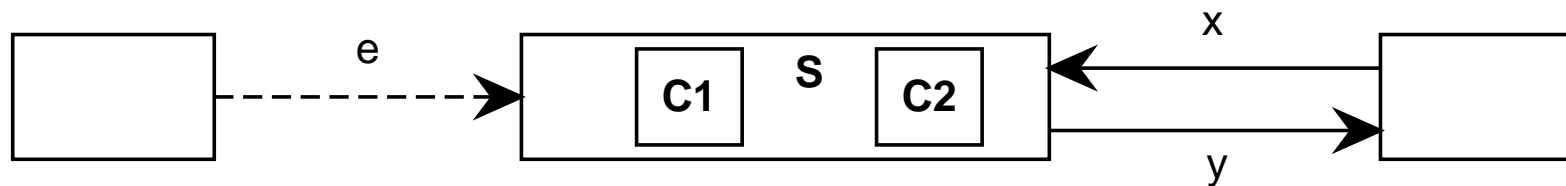
There are two kinds of addressing.

- **Channel addressing.** Item is sent to channel. Used in DFDs.
- **Destination addressing.** Item is sent to individual destinations. Used in the UML.

(De)composition

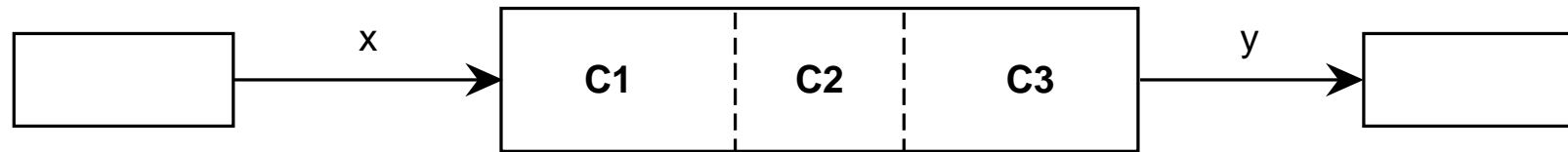
Represented by

- containment of nodes
- or simply specifying the component elsewhere.



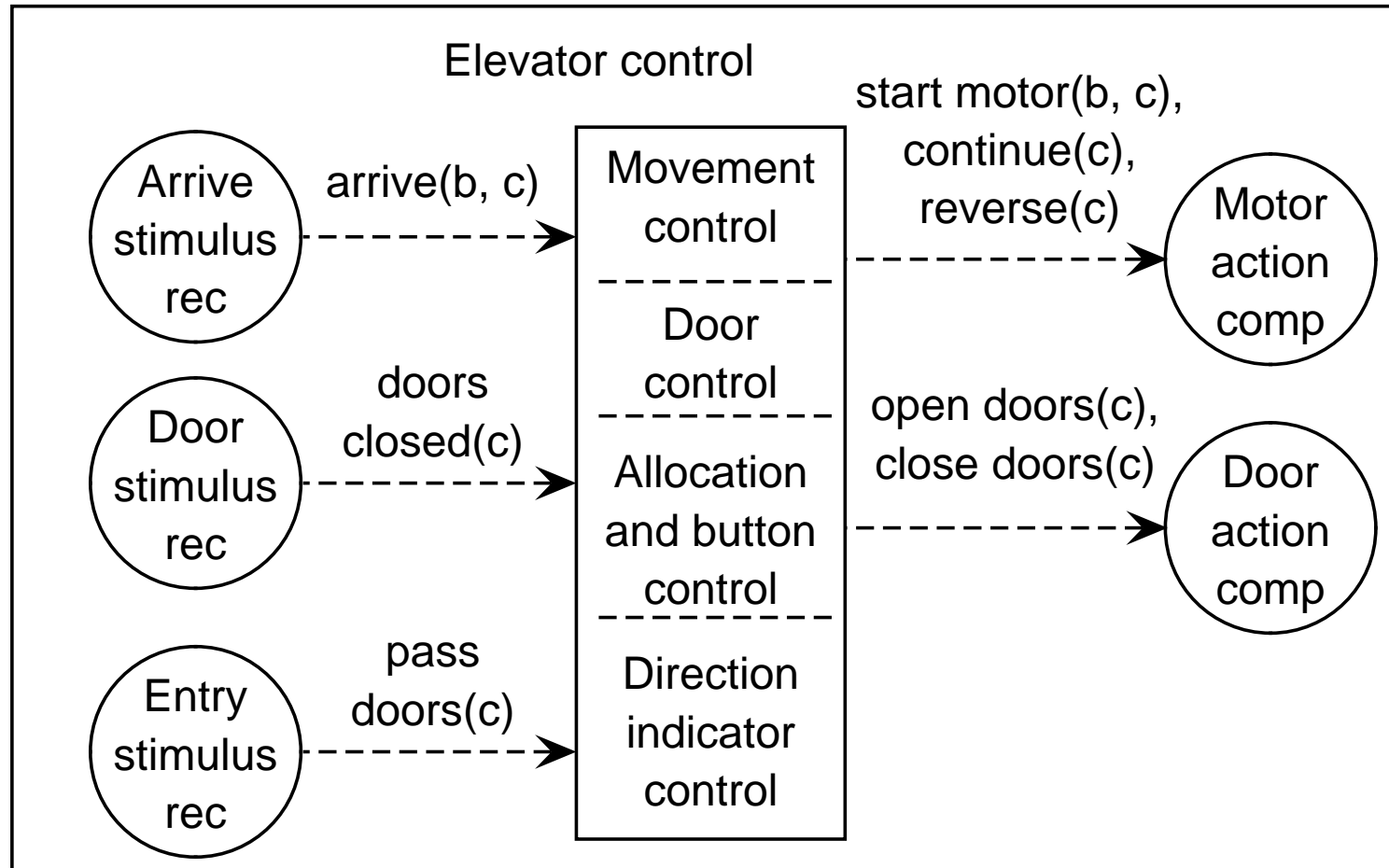
- C1 and C2 can both be triggered by an occurrence of event e.
- C1 and C2 can both read the value of x.
- C1 and C2 can both write a value to y.

Closely coupled components



- C1, C2 and C3 have same interface.
- Events generated in one component are sensed by the other closely coupled components. (Event broadcasting)
- The state of any closely coupled component is readable to any other closely coupled component. This permits the use of in(State).

Elevator controller composition fragment



Arrows show possible interfaces of all four closely coupled components.

Allocation of services to components

Compo- nents	Functions				
	Create batch data	Start heating	Switch on heater	Switch off heater	Finish heating
Enter batch data	X				
Batch data	X	X			
Start heating		X			
Tank control			X	X	
Monitor end time					X

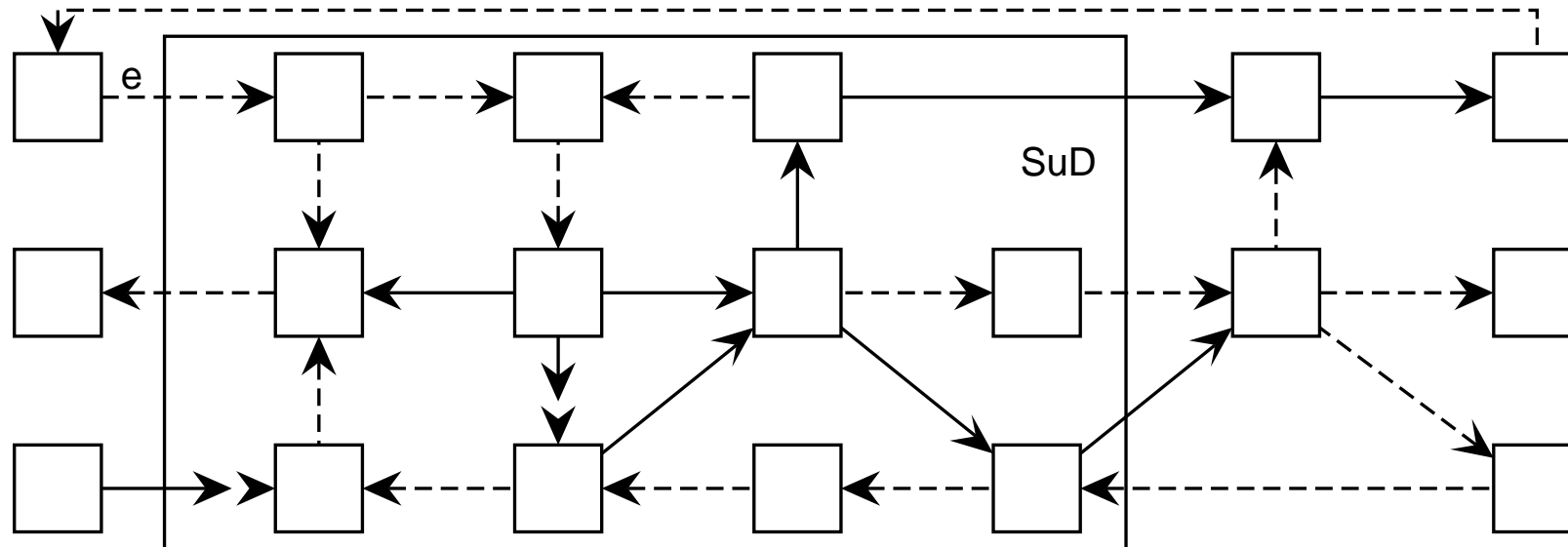
Flowdown

Compo- nents	Functions				
	Create batch data	Start heating	Switch on heater	Switch off heater	Finish heating
Enter batch data	Accept and store data.				
Batch data	Store data	Provide data			
Start heating		Start heating			
Tank control			Switch on when too cold	Switch off when too hot	
Monitor end time					Stop when time is up.

Main points

- Communication diagrams generalize DFDs.
- They allow us to represent objects and their classes in the diagram, and to represent close coupling.
- Instance-level diagram represents snapshot of the system.
- Decomposition can be represented by containment of nodes.
- We must allocate and flow down system services to component services.

Chapter 17. Communication Semantics



What is the response to e ?

- Depends upon the behavioral semantics of the components.
- We assume the semantics of component specifications is fixed, and concentrate on the remaining choices.

Input buffers: problem and possible solutions

Most combinations of semantic choices require a component to deal with a backlog of events that occurred but have not yet been responded to. Suppose components have **input buffers**.

- Size of buffer?
- Structure of buffer? (Set, bag, queue)
- Removal policy. (Even for queues, i.e. deferred events.)

State: set of unlimited size.

UML: Queue (more or less) of unlimited size.

Component input and output: problem and possible solutions

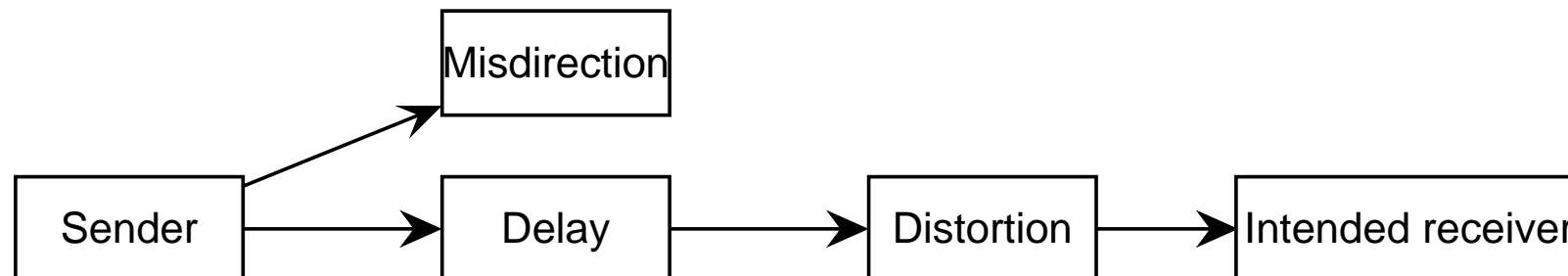
Computations during a step may need more input, or may produce output.

- **Eager read/lazy write:** All input values are read eagerly, at the start of the step, and all output (actions and data) is written lazily, at the end of the step. Statemate.
- **Lazy read/eager write:** Input is read lazily, when needed, and output is written eagerly, immediately when available. UML.

Semantics of communication channels

- There is no delay between sending and receiving,
- A message always arrives at its destination(s),
- It arrives at no other destinations and
- A message is never distorted.

We can always introduce these things explicitly:



Addressing mechanisms

- **Channel addressing.** “Deliver this communication to all components at the other side of this channel.” DFDs and Statemate. Good for broadcast, bad for point-to-point communication; good for encapsulation.
- **Destination addressing.** “Deliver this message to this component.” Good for point-to-point, bad for encapsulation. UML.

Channel capacity: semantic options

- **Zero channel capacity:** Message immediately arrives at receiver. If the receiver has no input buffer, a channel represents a **synchronous communication** in which the sender cannot put an item in a channel if the receiver does not take it out at the same time.
- **n -item channel capacity:** A channel can contain n items at the same time. When a sender wants to write something to a channel that is full:
 - **Overwrite semantics.** (Statemate and UML)
 - **Refusal semantics.**

Blocking: semantic options

What happens when for some reason, a sender attempts a communication through a channel, but this communication is not possible?

- **No blocking:** Discard immediately.
- **Finite blocking:** Wait a finite time, then discard. else.
- **Infinite blocking:** Wait.

Statemate and UML: Not applicable, because they use overwrite semantics.

Temporal semantics of network behavior

The SuD is a network of communication components.

Options for temporal semantics:

- Global time is indicated by global clock, immediately accessible to all components.
- Local clocks may differ.

Step semantics of network behavior

- **Network step semantics.** The SuD is ready to respond when any of its components is ready.
- **Network multistep semantics.** The SuD is ready to respond when all its components are ready.

Propagation strategy:

- **Breadth-first:** Execute a triggered component after all components triggered earlier, are executed.
- **Depth-first:** Execute a triggered component immediately.

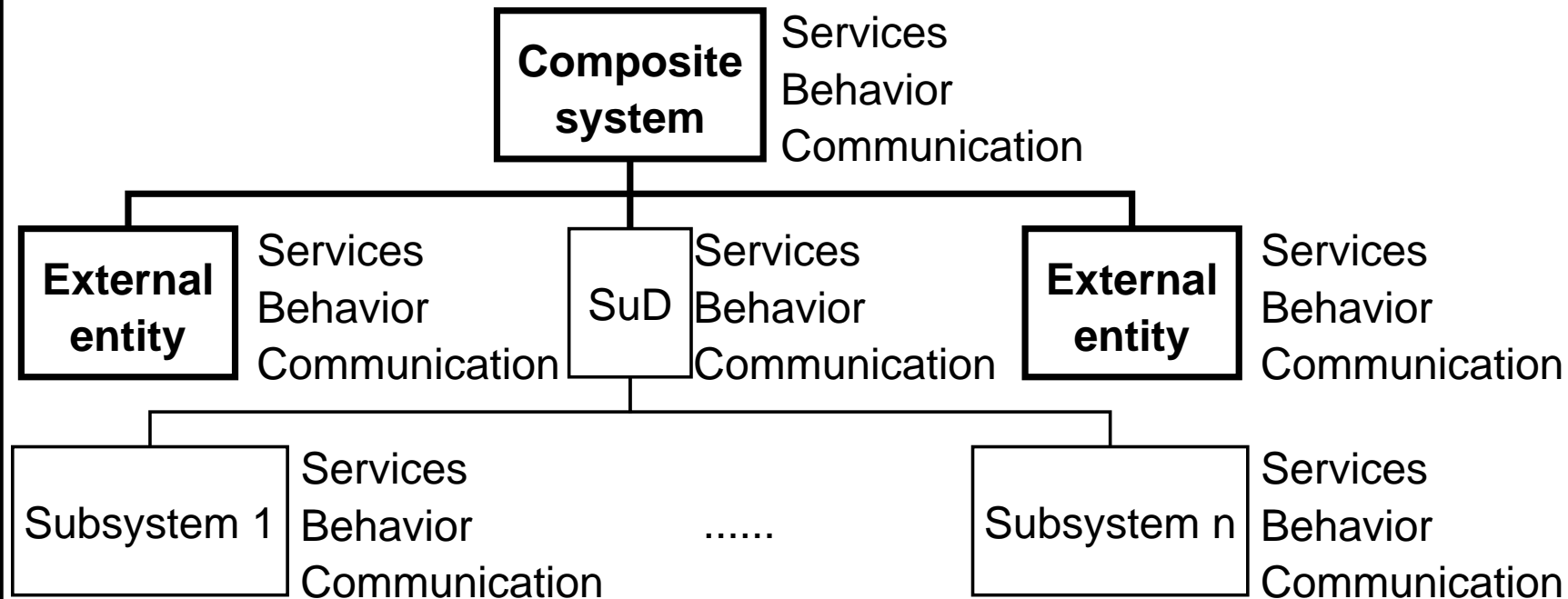
The Environment

- A communication diagram is not intended to specify environment behavior. But it does make some assumptions about environment behavior.
 - Environment is able to absorb response at all times.
 - The environment can produce stimuli at all times.
- The environment is continuous, the system is discrete.
 - Arrival order of stimuli over different channels may be unknown by the SuD.
 - Stimuli arriving over same channel may be lost.
 - Time-continuous input is sampled.

Main points

- Components may have input buffers; may have size restrictions and access policies.
- Communication channels are reliable and instantaneous.
- Addressing may be by channel or by destination.
- Channel capacity may be restricted.
- Communications have a blocking semantics.
- Time may be global or local.
- Steps may propagate through the network breadth-first or depth-first.
- We assume that the environment is always able to absorb responses.

Chapter 18. Context Modeling Guidelines



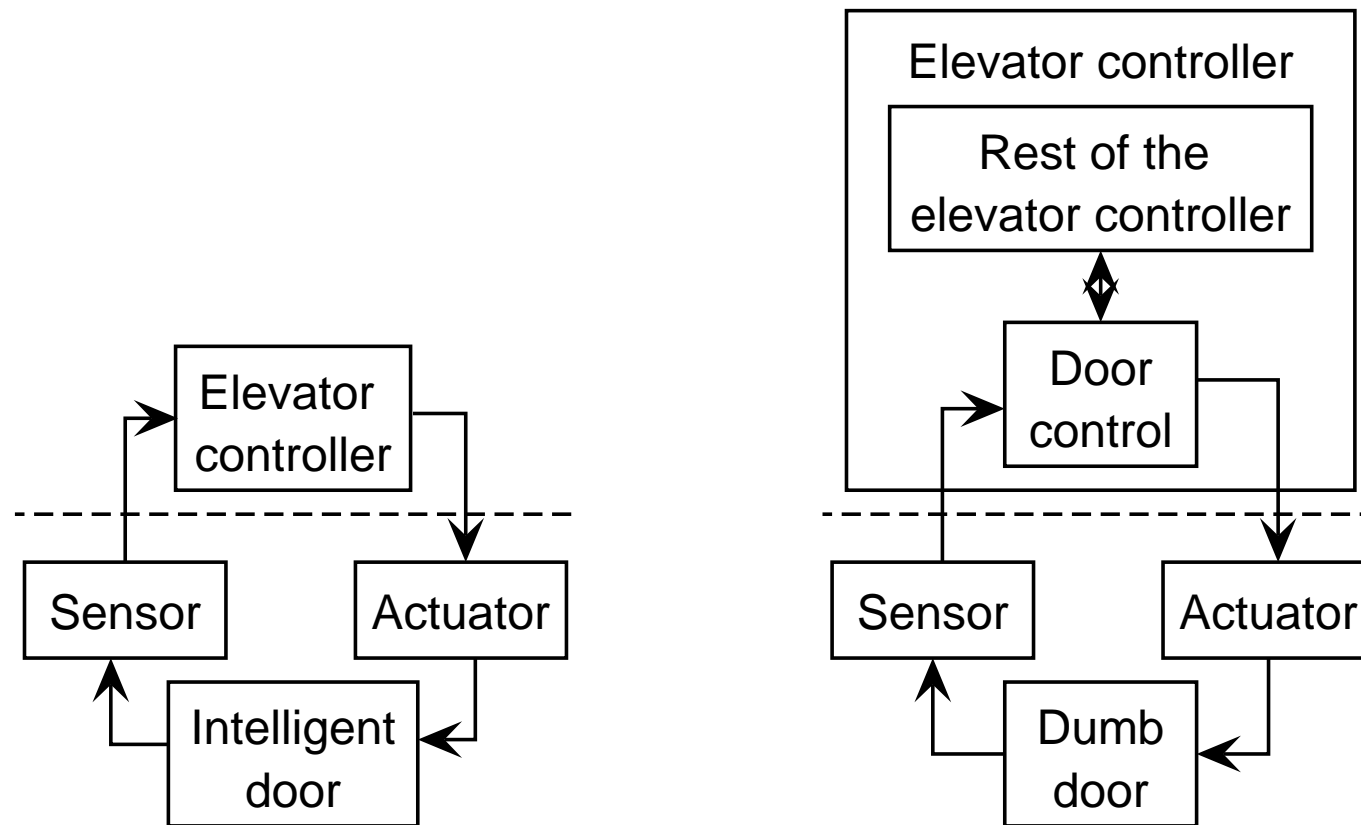
- To prepare for making design decisions, you first collect information about things given to you.
- This is modeling, not design.
- Consider system as black box.

Where is the system boundary?

- ✓ Use function refinement tree. This is a fully implementation-independent description of system boundary.
- ✓ Use list of stimuli and responses. This is a behavior description of the system boundary.

Tradeoff between environment and SuD

- System engineering argument: A and S entail E .
- Responsibility for emergent properties is distributed over environment (A) and system (S).



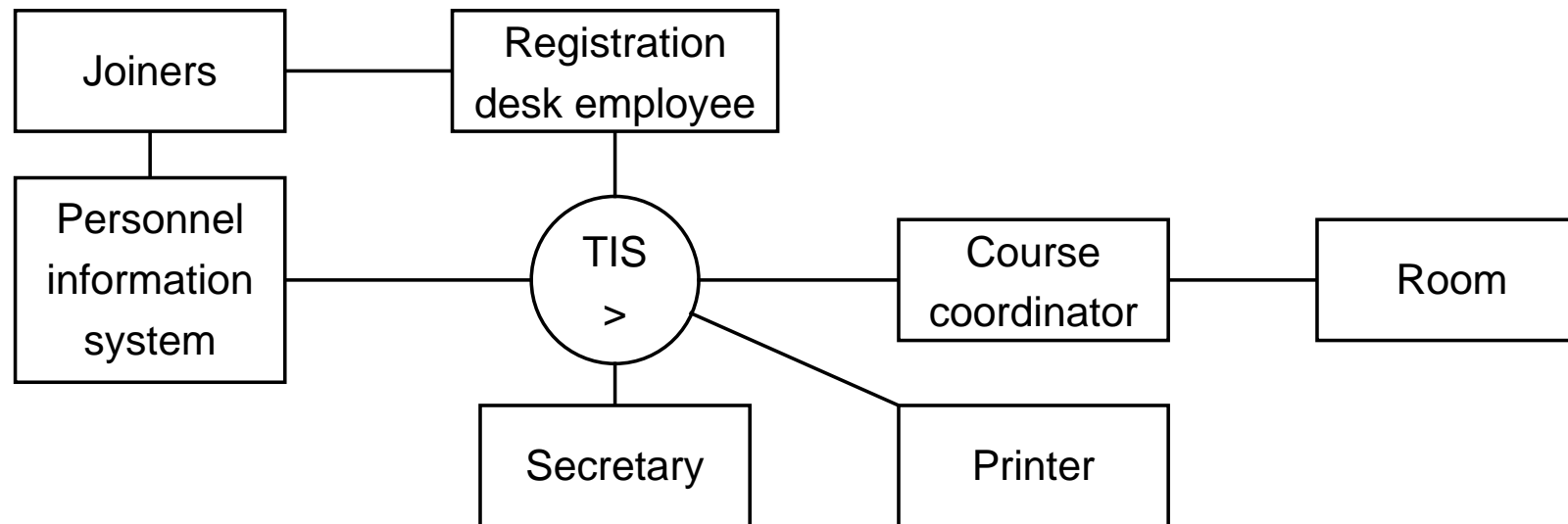
Context diagram

Represents communication among the relevant entities in the environment and the SuD.

- ✓ External entities are
 - Physical entities (people, devices, natural objects),
 - Conceptual entities (organizations, obligations, rights),
 - Lexical entities (software, contracts, specifications).

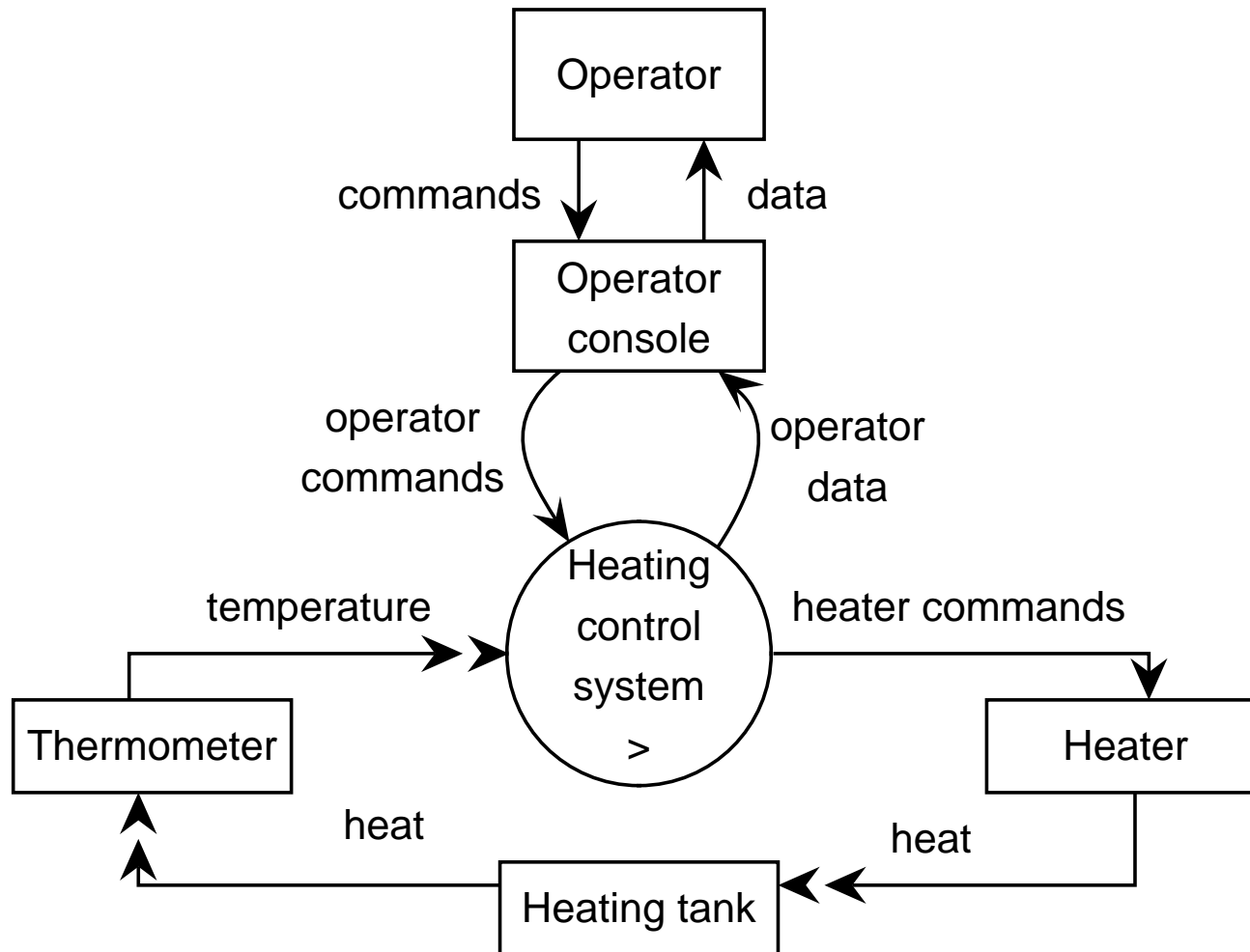
A context diagram must give overview; it should link clearly to the purpose of the system.

Teaching information system



Not much detail needed here.

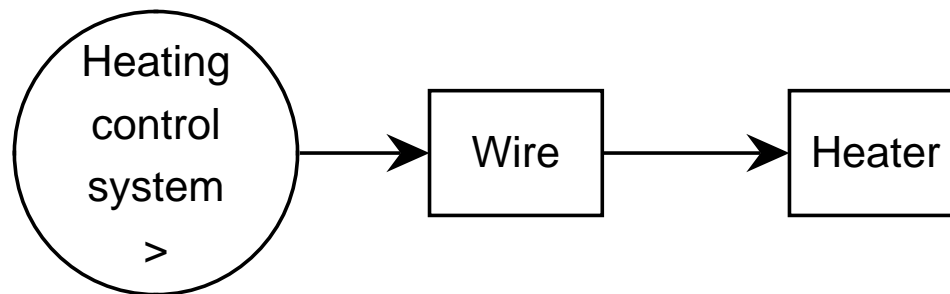
Heating controller



More detail needed here.

Choose an abstraction level

✓ Communication channels represent a level of abstraction.



There is always a remaining level of abstraction.

Context boundary

- ✓ Include entities needed to achieve goals of composite system.
- ✓ Include entities in which SuD causes desired effect.
- ✓ Include entities about which SuD needs information to perform its work.
- ✓ Ignore entities where effect is not felt / not relevant.
- ✓ Ignore entities whose behavior is irrelevant for the task of the SuD.

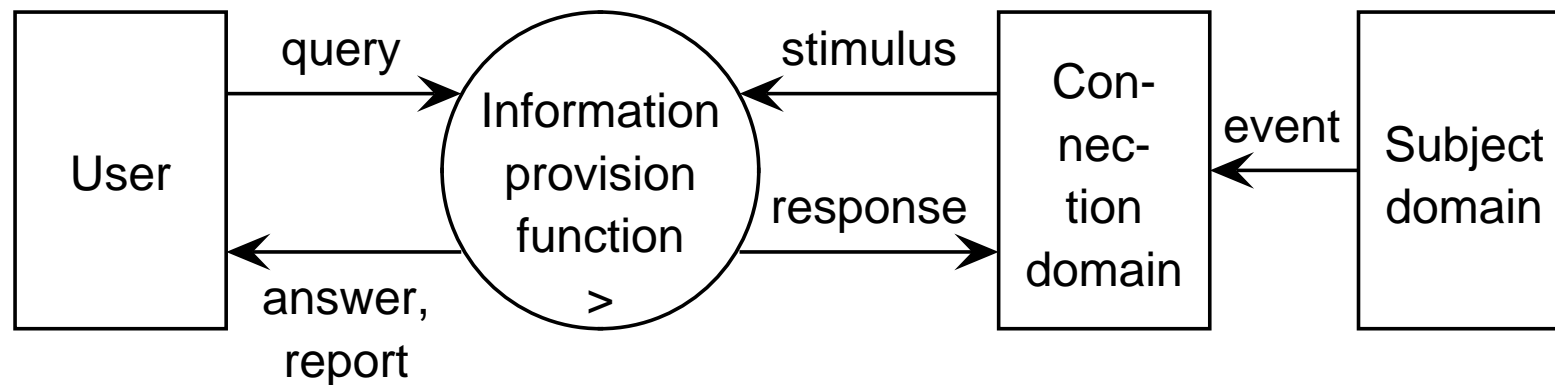
The context diagram shows which external entities are, together with the SuD, involved in achieving the desired emergent properties of the composite system.

Structuring the context

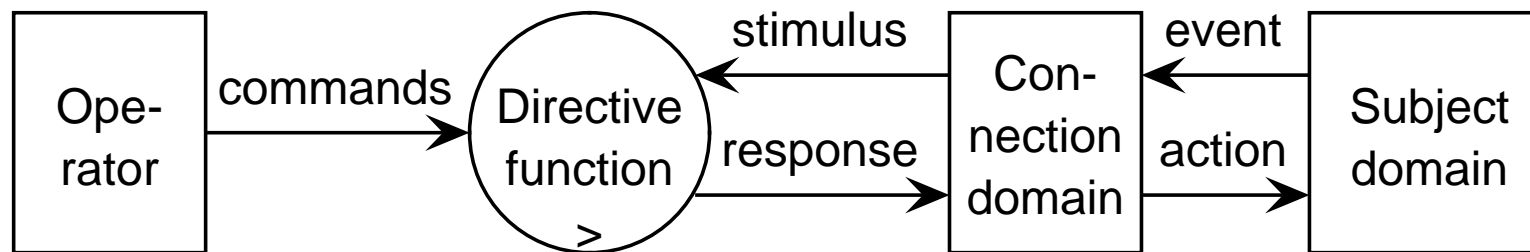
Three kinds of functionality:

- *Information provision.* To answer questions, produce reports, or otherwise provide information about the subject domain.
- *Direction.* To control, guide, or otherwise direct its subject domain.
- *Manipulation.* To create, change, display, or otherwise manipulate lexical items in the subject domain.

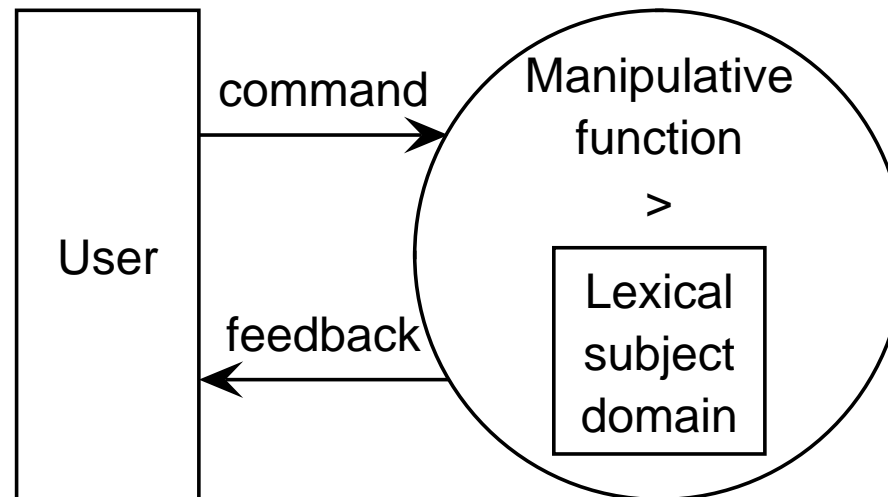
Typical structure in the context of an information-provision system



Typical structure in the context of a directive system



Typical structure in the context of a manipulative system

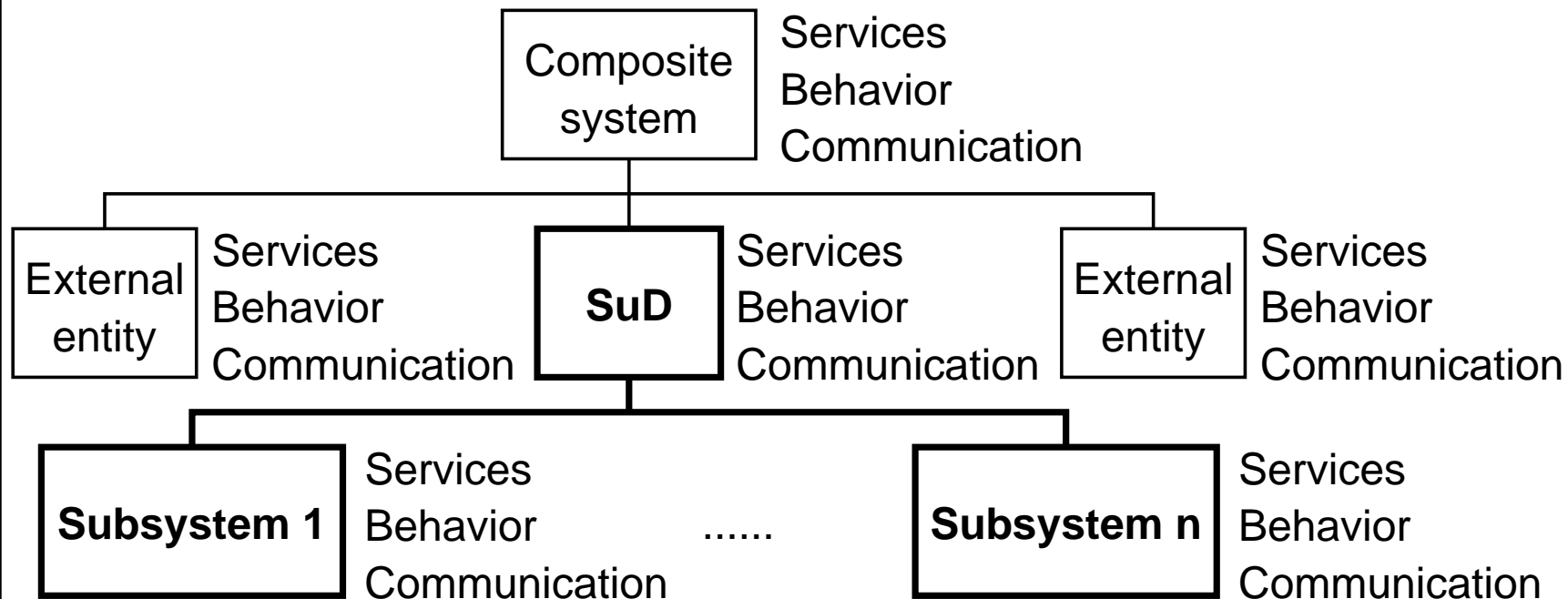


Main points

- Context is modeled, not designed.
- System boundary is determined by desired system services.
- Trade-off between functionality in the system and functionality in the environment.
- Context ends where relevant effects or relevant information ends.
- Context models have typical structures that depend upon typical system functionality.

Chapter 19. Requirements-Level Decomposition Guidelines

This is design, not modeling.

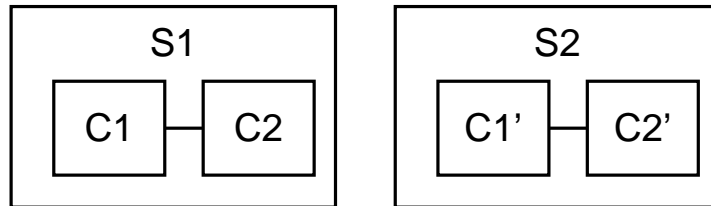


Architecture and architectural style

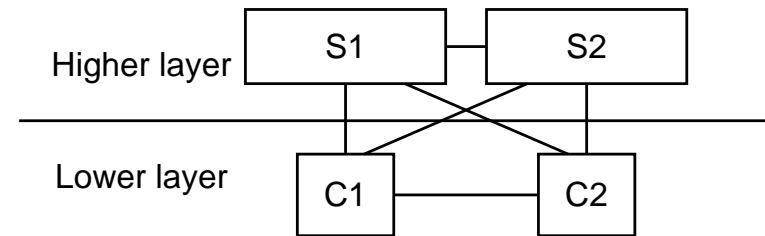
Architecture = a structure of elements and their properties, and the interactions between these elements that realize system-level properties.

- A system can have many architectures: e.g. requirements-level, implementation-level, execution-level, code-level.
- Elements in an architecture must have synergy. They must jointly produce emergent properties.
- An **architectural style** is a set of constraints on an architecture.

Basic style choice: Decomposition versus layering

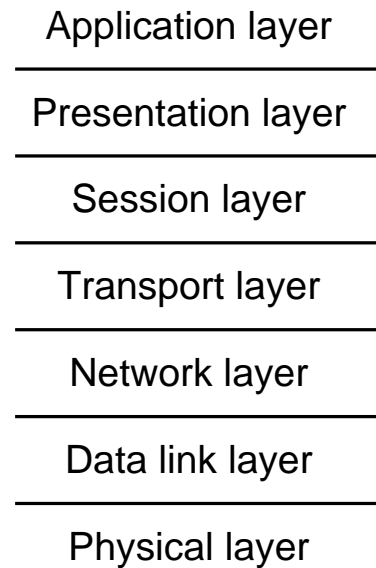


- C1, C2 deliver *services* to S1.
- C1', C2' deliver *services* to S2.
- S1 *encapsulates* C1, C2.
- S2 *encapsulates* C1', C2'.

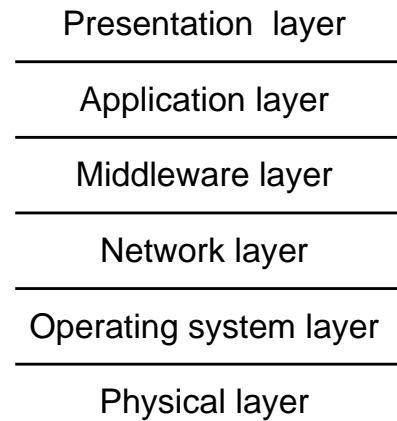


- C1, C2 deliver *services* to S1 and S2.
 - C1, C2 *are not encapsulated* by S1 or S2.
-
- Layering and decomposition can be mixed at various levels.
 - Layering can be strict or loose.

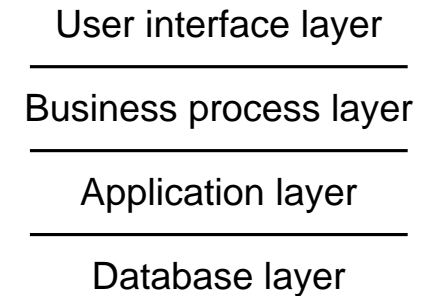
Examples of layered contexts



(a)

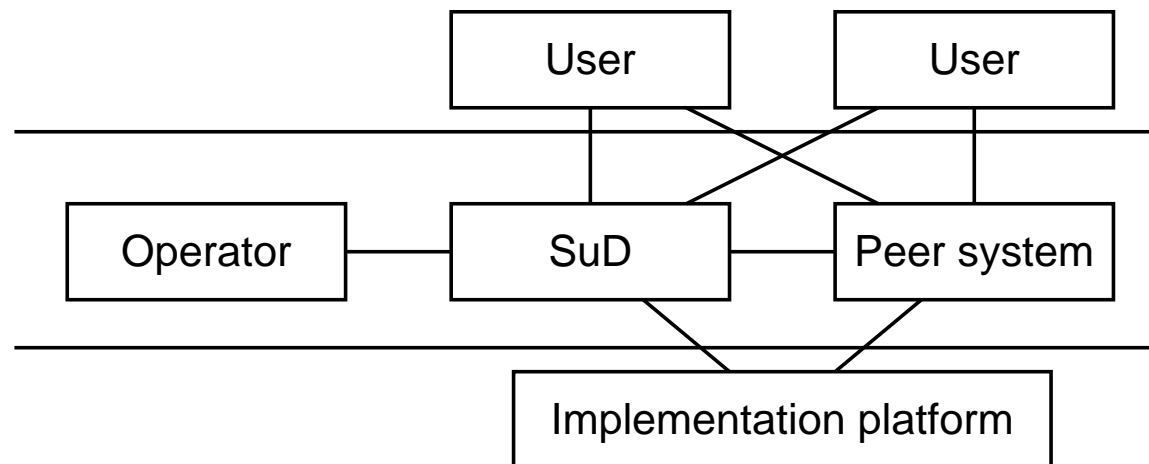


(b)



(c)

Layered context diagram

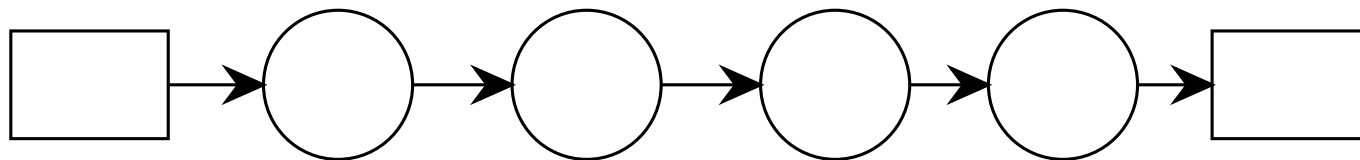


Structuring guidelines

- ✓ Choose a context structure that reflects the problem structure.
- ✓ Choose an SuD architecture that localizes changes.
 - Keep related data together
 - Keep related functions together

Architectural styles (1)

Data flow style:

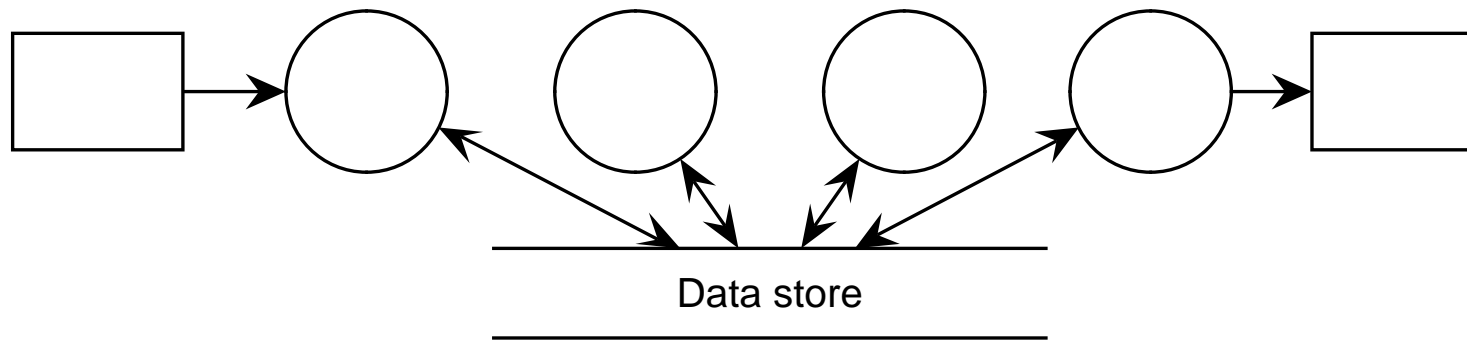


- Pure data flow style: data flows through a network of transformations (that have no persistent memory).
- Many variants: E.g. batch, pipe-and-filter.

Not applicable to reactive systems, because these need a (persistent) model of their environment.

Architectural styles (2)

Von Neumann style:



- This is the classical information system architecture style: Databases and application programs.
- Variant: *Blackboard style*: Intelligence in the data store(s). These can announce that they have been updated.

Architectural styles (3)

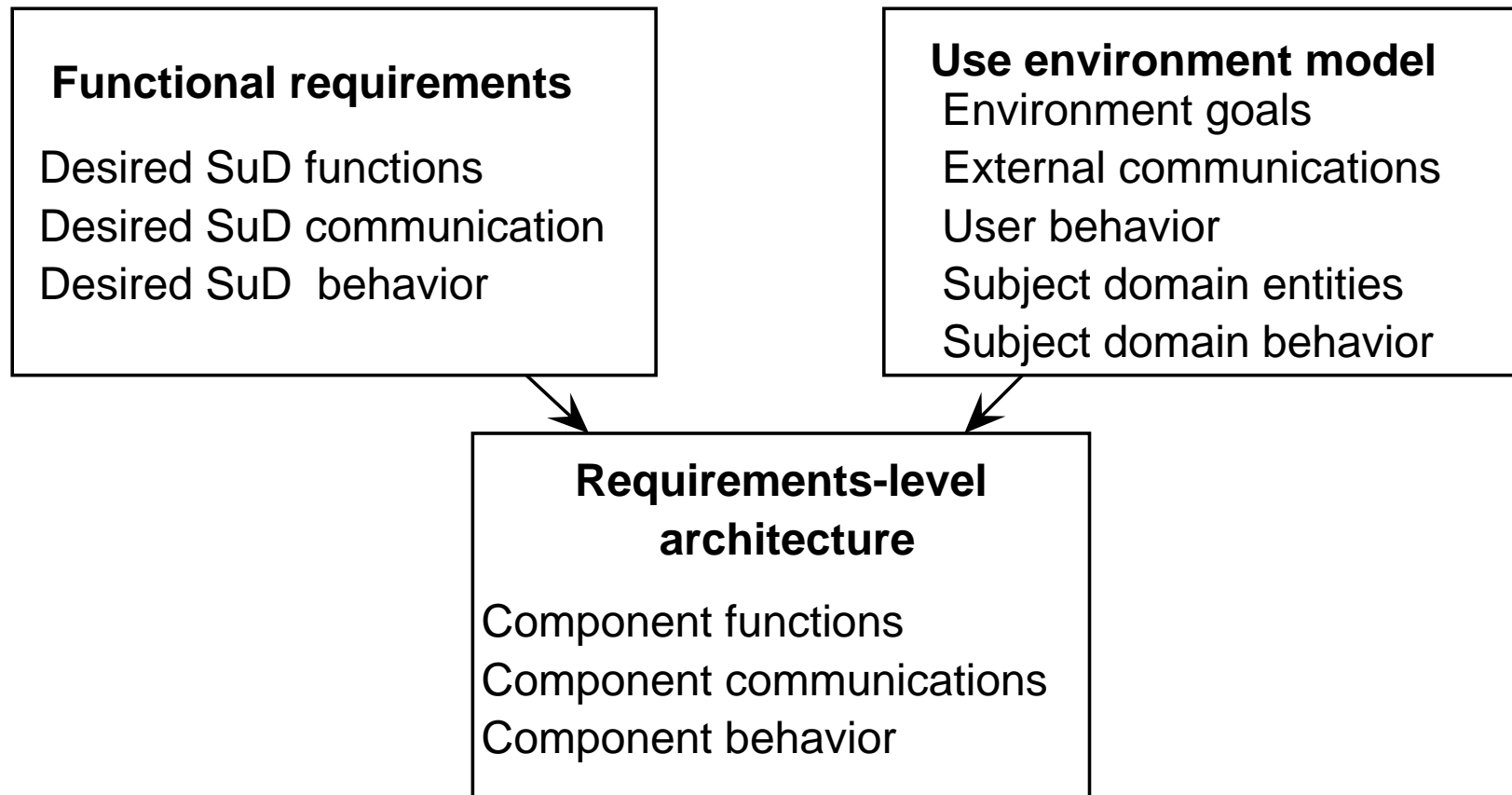
Object-oriented style.

- All components are objects.
- Usually with destination addressing.
- Variant: *publish-subscribe style*: kind of dynamically configurable channel addressing.

Requirements-Level Architecture Design Approach

- **Requirements-level decomposition:** Defined only in terms of requirements and environment models.
- Also called “conceptual architecture” or “logical architecture” or “essential system model”.
- Requirements-level decomposition assumes “perfect technology” because it ignores technology. It is a restatement of the requirements in terms of a decomposition.
- **Implementation decomposition:** Mapping of requirements-level decomposition to some implementation platform.

Sources of design decisions



Classification of design decisions

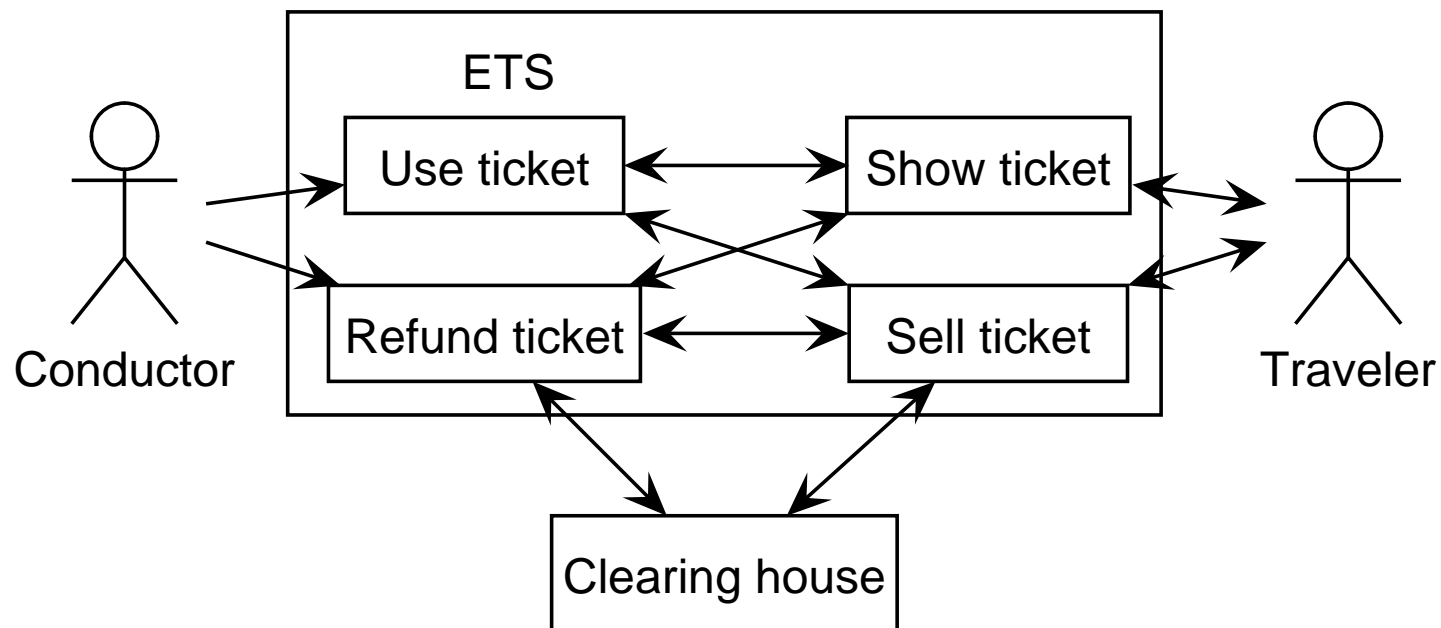
Design decisions for the requirements-level architecture are made in terms of

- Functions
- External communication
 - Events
 - Devices
 - Users
- External behavior
- Subject domain structure

Design guidelines

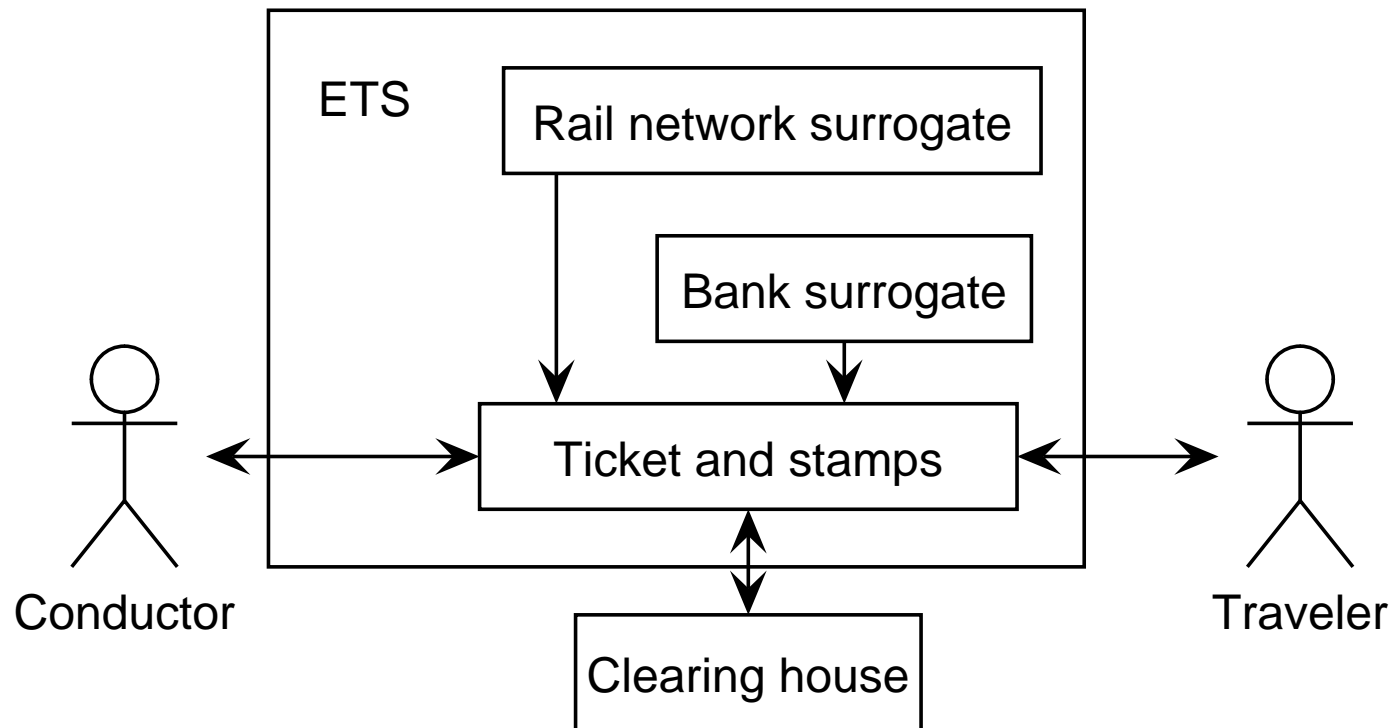
- ✓ Using **functional decomposition**, each system function is allocated to a different component.
- ✓ Using **subject-oriented decomposition**, each group of subject domain entities corresponds to a system component.

ETS example (1): Pure functional architecture —object-oriented style



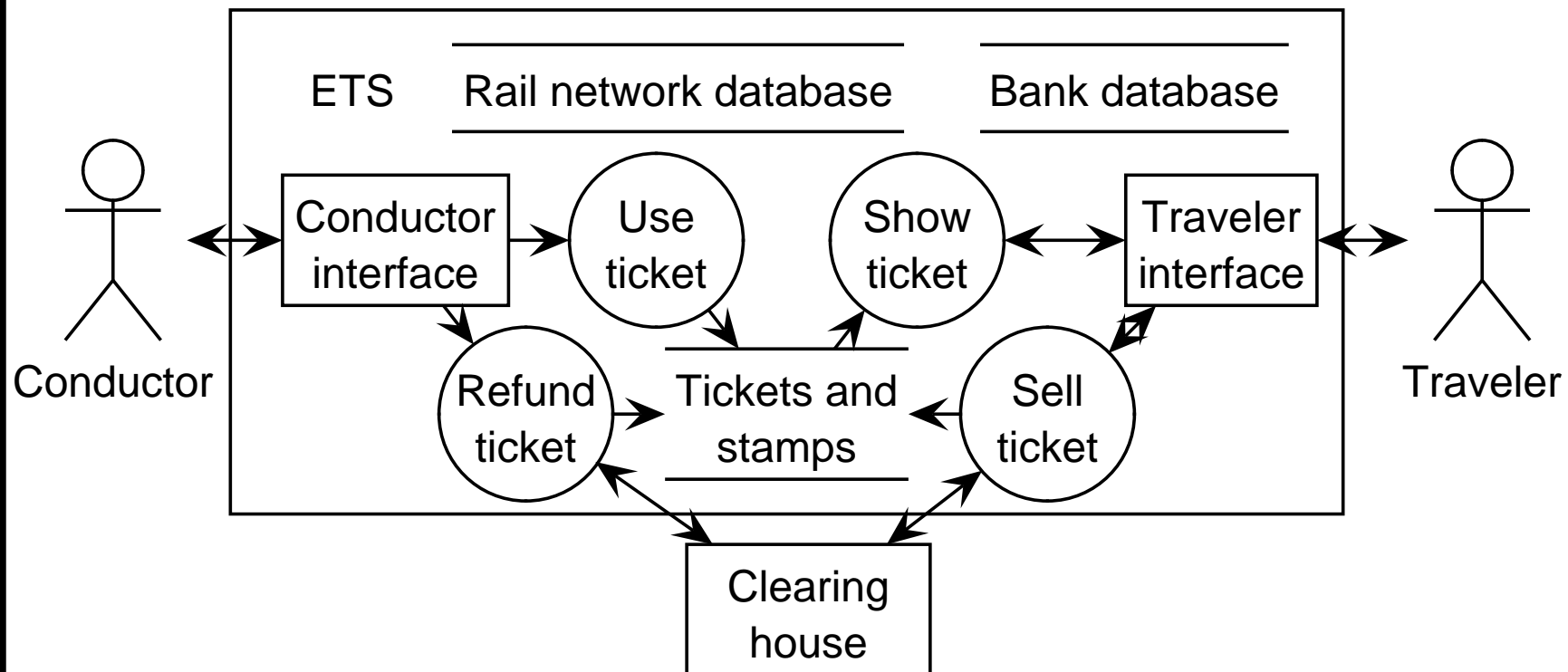
- All data encapsulated with functions!
- Inefficient.
- Does not adequately represent problem structure.

ETS example (2): Pure subject-oriented architecture —object-oriented style



- All functions encapsulated in data!
- Inefficient.
- Does not adequately represent problem structure.

ETS example (3): Mixed architecture — Von Neumann style



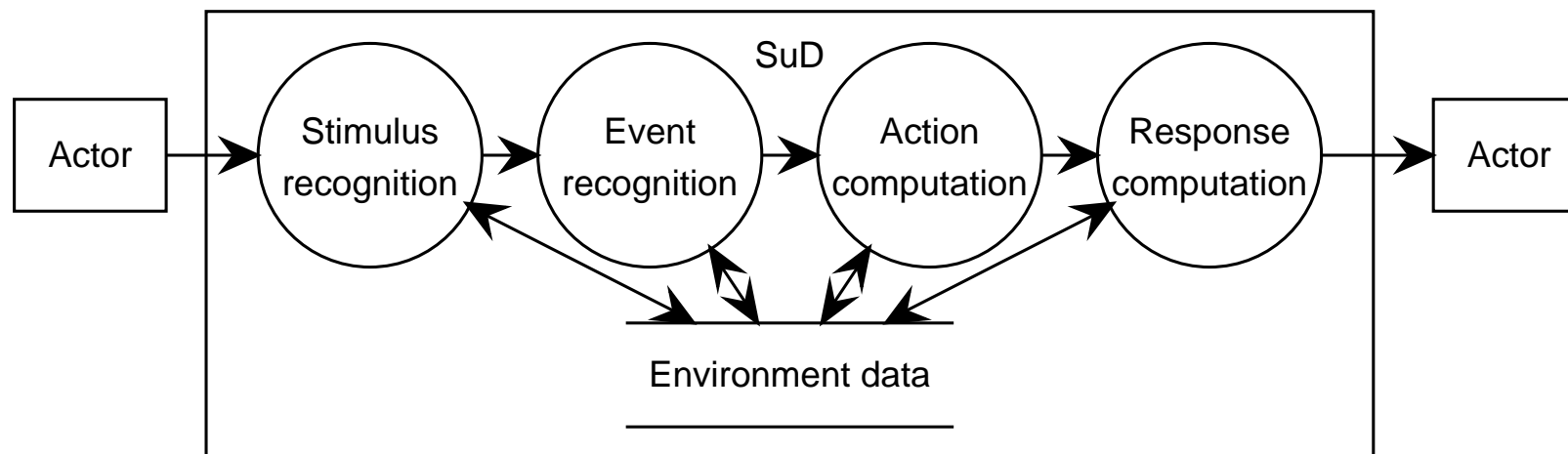
Criteria used:

- Functional decomposition
- Subject-oriented decomposition

- User-oriented decomposition

Communication-oriented decomposition

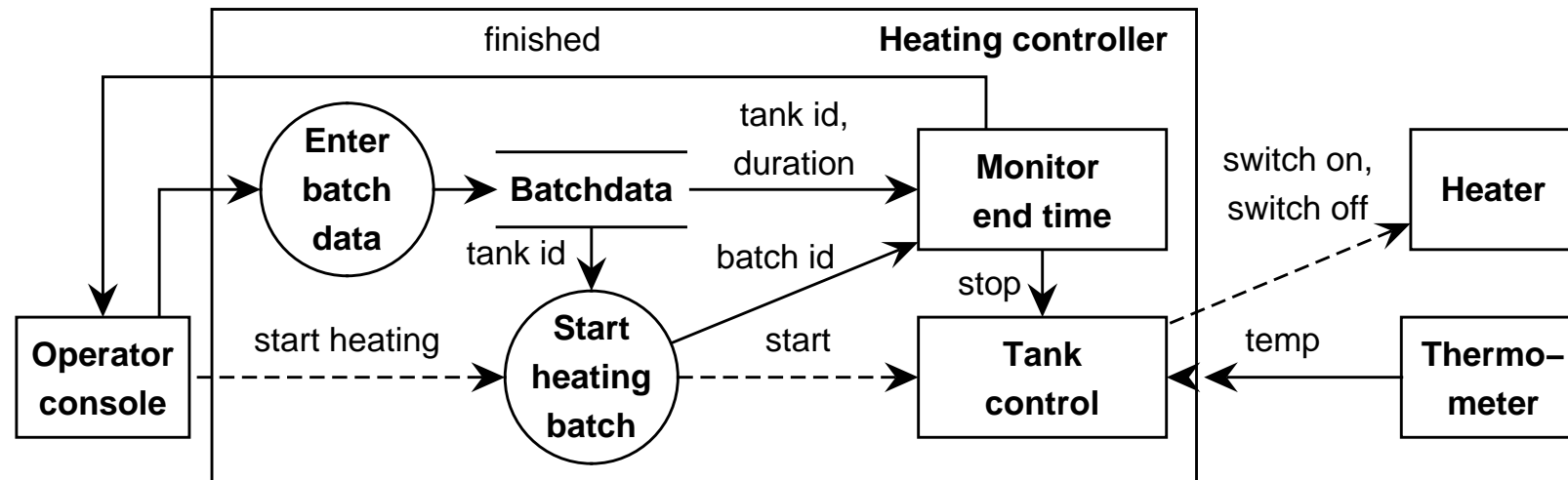
- ✓ Using **event-oriented decomposition**, each event is handled by a different component.
- ✓ Using **device-oriented decomposition**, each device is handled by a different component.
- ✓ Using **user-oriented decomposition**, communications with one kind of user are handled by one component.



Behavior-oriented decomposition

- ✓ In **behavior-oriented decomposition**, monitoring assumed behavior in the environment, or enforcing desired behavior in the environment, is allocated to one component.

Heating controller example (1): Mixing functional and subject-oriented decomposition

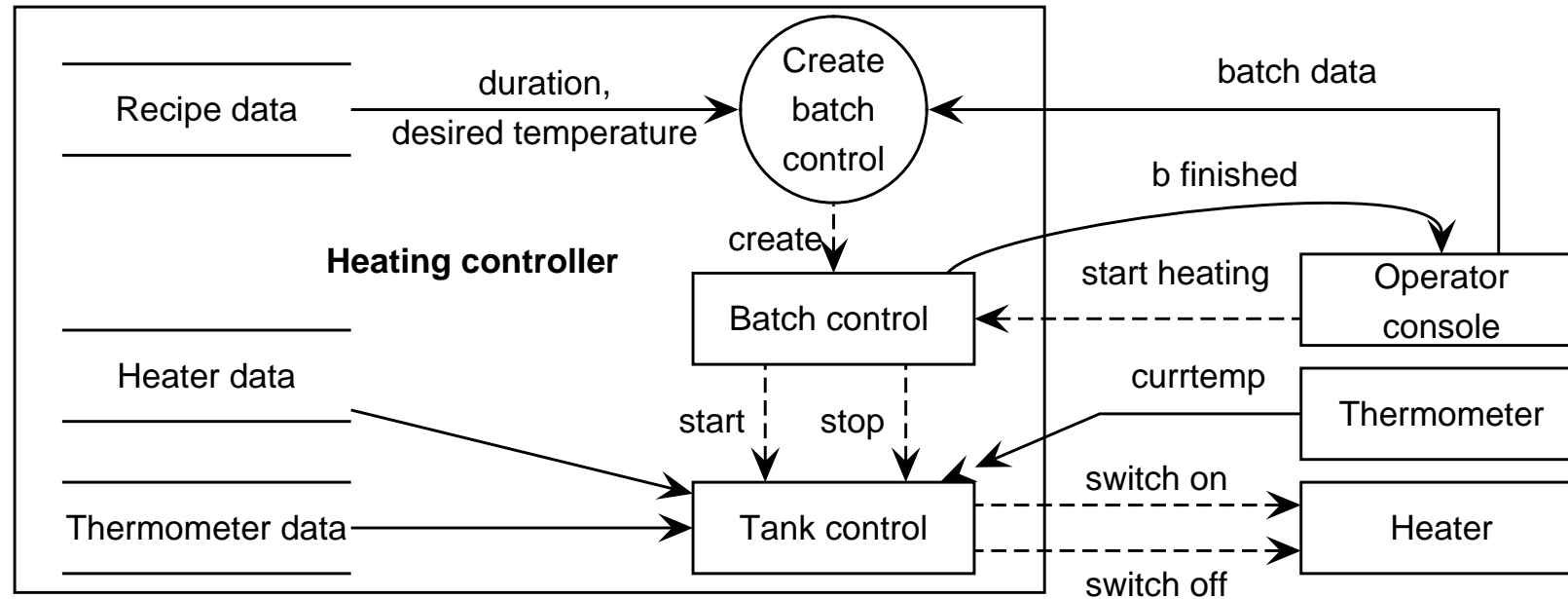


Guidelines used:

- Functional
- Subject-oriented

Different aspects of batch behavior are left scattered around.

Heating controller example (2): Including behavior-oriented decomposition



Guidelines used:

- Functional
- Subject-oriented
- Behavior-oriented

Evaluation criteria

- ✓ Check that all data used is created, that all data created is deleted.
- ✓ Execute the model.
- ✓ Produce a correctness argument that C_1 and ... and C_n entail S .
- ✓ Check that quality attributes (efficiency, safety, reliability, etc.) are realized.
- ✓ Build a throw-away prototype and experiment with it.

Main points

- Distinguish requirements-level decomposition from implementation decomposition.
- Basic architectural choice is between layering and encapsulation.
- Other major architectural styles: Data flow, Von Neumann, Object-oriented.
- Requirements-level decomposition guidelines:
 - Functional,
 - Subject-oriented,
 - Communication-oriented: events, devices, users
 - Behavior-oriented

These correspond to major system aspects & subject domain.

Chapter 20. Postmodern Structured Analysis (PSA)

History of structured analysis;

- Introduced in the 1970s as requirements specification method.
- Abstracts structured programming to requirements level.
- Idea (1): SuD structure should match problem structure rather than structure of implementation platform.
- Idea (2): The SuD should be modular.

Yourdon-style structured analysis claims that modularity is achieved by functional decomposition. But:

- Functional decomposition is not modular if there are many interfaces between the functions.
- Chapter 19 (Requirements-Level Architecture Guidelines) lists many other decomposition guidelines.

Postmodern structured analysis (PSA)

Differs from classical structured analysis:

- ERD used for subject domain only.
- Extended context diagram.
- Event flows may contain data.
- STDs can be statecharts.
- STDs may have local variables.

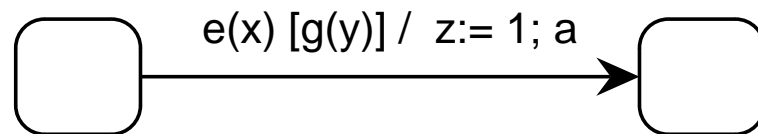
Notations used in PSA.

Design level	Notation
Environment	<ul style="list-style-type: none"> • Context diagram • ERD of subject domain • Event-action lists of desired subject domain behavior • Event-action lists of assumed subject domain behavior
Requirements	<ul style="list-style-type: none"> • Mission statement • Function refinement tree • Service descriptions • Stimulus-response list of desired system behavior
SuD decomposition	<ul style="list-style-type: none"> • DFD SuD decomposition • STTs or STDs of control processes
	<ul style="list-style-type: none"> • Dictionary

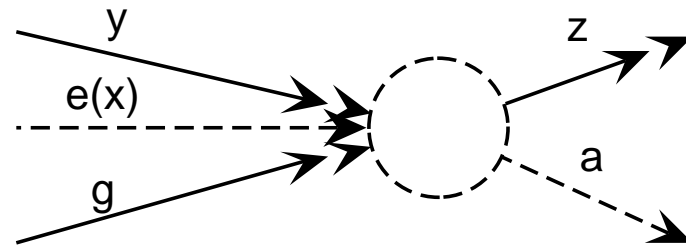
Coherence rules

- *Environment models.*
 - Context diagram shows relevant communication paths between SuD and subject domain.
 - Event-action pairs refer to subject domain entities involved.
- *Requirement specifications.*
 - Mission statement = root of function refinement tree.
 - Service descriptions = leaves of FRT.
 - Each function is triggered by stimulus in SR list.
 - Each stimulus-response pair is part of a function.
- *Decomposition specifications.*
 - Each control process is specified by a behavior description.
 - Each behavior description describes a process in the DFD.

Relation between an STD and the control process that it specifies



(d)



(e)

Coherence across models

- *Environment and requirements.*
 - Event \rightsquigarrow stimulus and response \rightsquigarrow action ...
 - and each of these is a path in the context diagram.
- *Requirements and decomposition.*
 - For each SR pair, there is a path through the DFD.
- *Decomposition and environment.*
 - Context diagram is abstraction from DFD.
- *Dictionary.*
 - The dictionary defines at least the relevant subject domain terms for entities and events.

Flyweight to heavyweight

The weight of professional boxers is classified according to the following scheme:

flyweight	≤ 112 pounds
bantamweight	≤ 118 pounds
featherweight	≤ 126 pounds
lightweight	≤ 135 pounds
welterweight	≤ 147 pounds
middleweight	≤ 160 pounds
heavyweight	> 160 pounds

We can learn two things from this classification:

1. Lightweight is not the lightest weight.
2. Heavyweights can be as heavy as they want.

Flyweight to heavyweight use of notations in PSA

Flyweight	Featherweight	Middleweight	Heavyweight
Context diagr	Context diagr	Context diagr	Context diagr
	ERD	ERD	ERD
			EA list
Mission stmt	Mission stmt	Mission stmt	Mission stmt
	Function rept	Function rept	Function rept
	Service descs	Service descs	Service descs
			SR list
		DFD	DFD
			Behavior descs
	Dictionary	Dictionary	Dictionary

Main points

- PSA is an extension of Yourdon structured analysis that uses all notations of this book except communication diagrams.
- PSA can be used with any desired level of weightiness.
- PSA can be used with any of the decomposition guidelines of chapter 19.

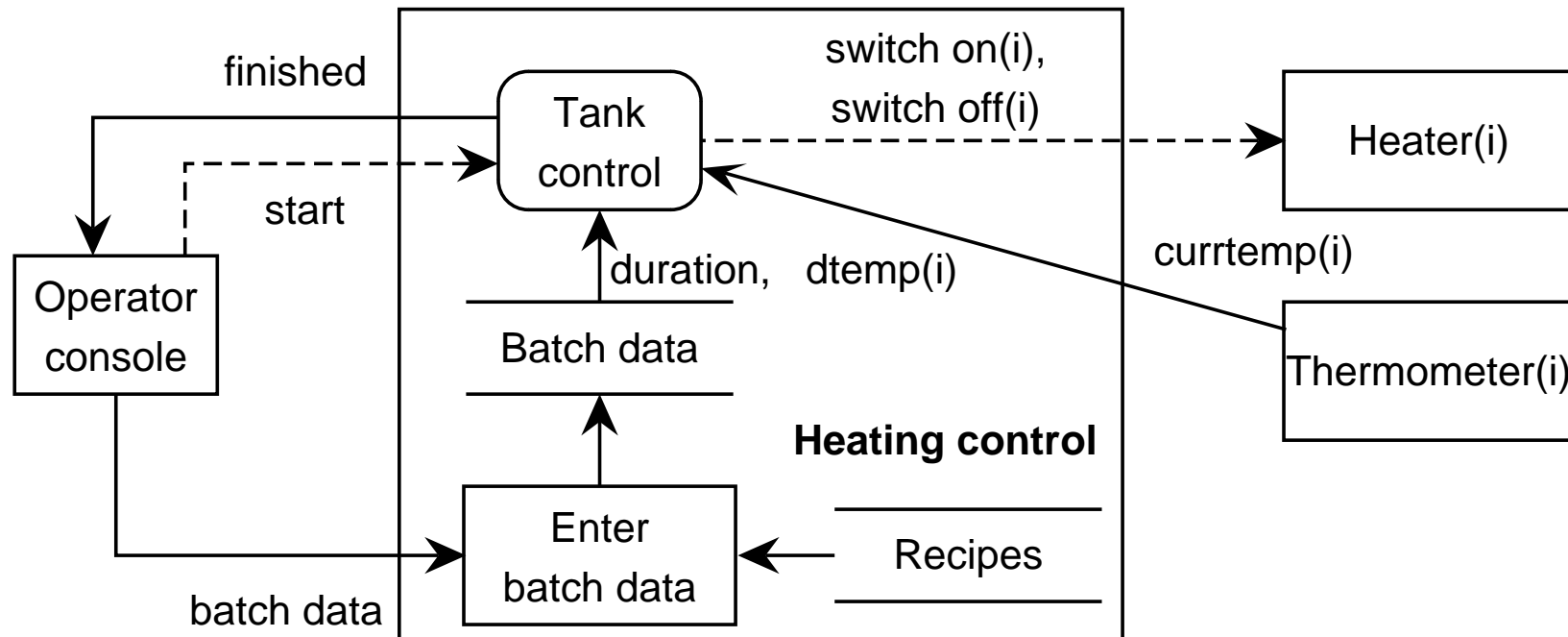
Chapter 21. Statemate

- Developed in the 1980s based on the idea of higraph:
Hierarchical hypergraphs.
 - Hyperedges.
 - Nodes can contain nodes, as in Venn diagrams.
- Activity charts are hierarchical DFDs.
- Control activities are specified by statecharts.
- There is a precisely defined execution semantics for activity charts and statecharts.
- Module charts represent computational resources (not treated here).

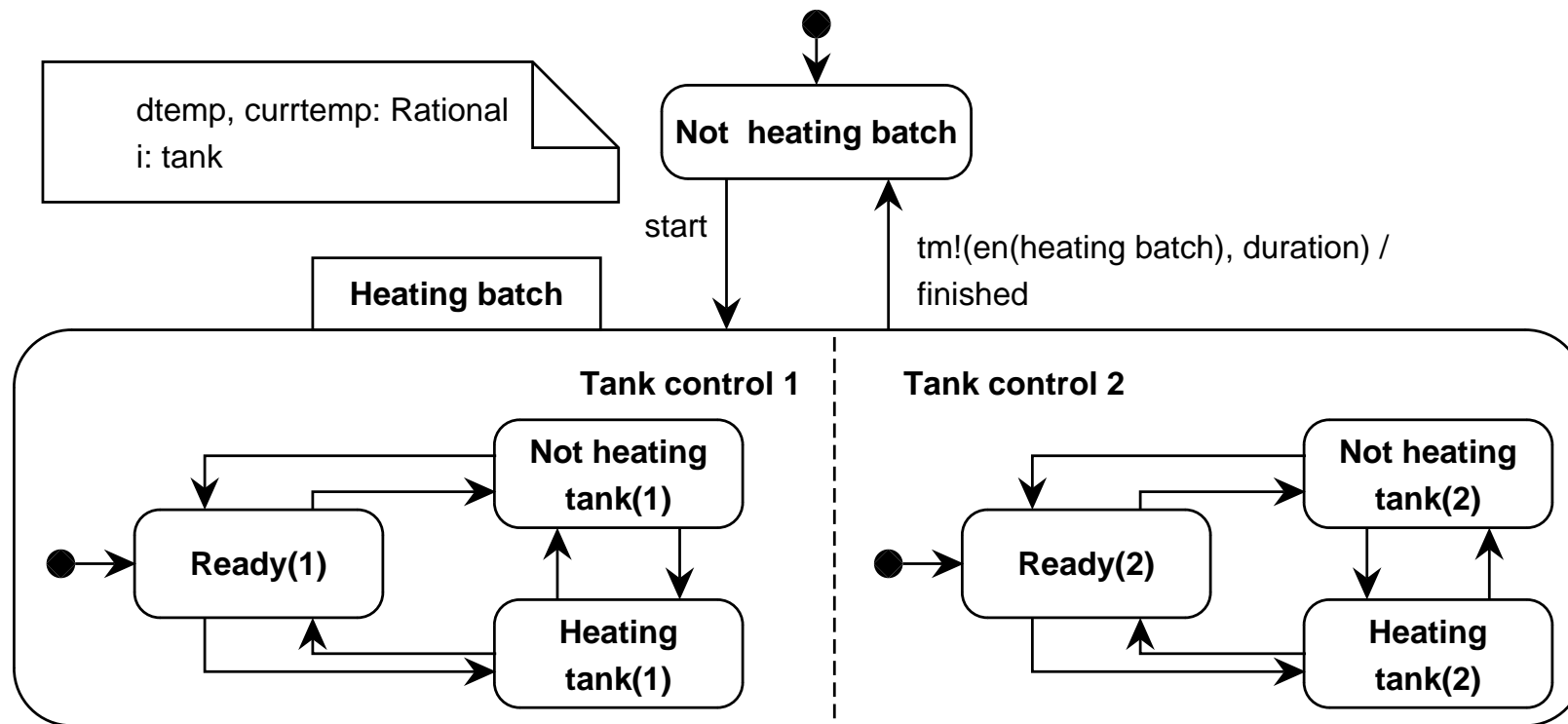
Statemate can be used in combination with other PSA notations.

Activity chart

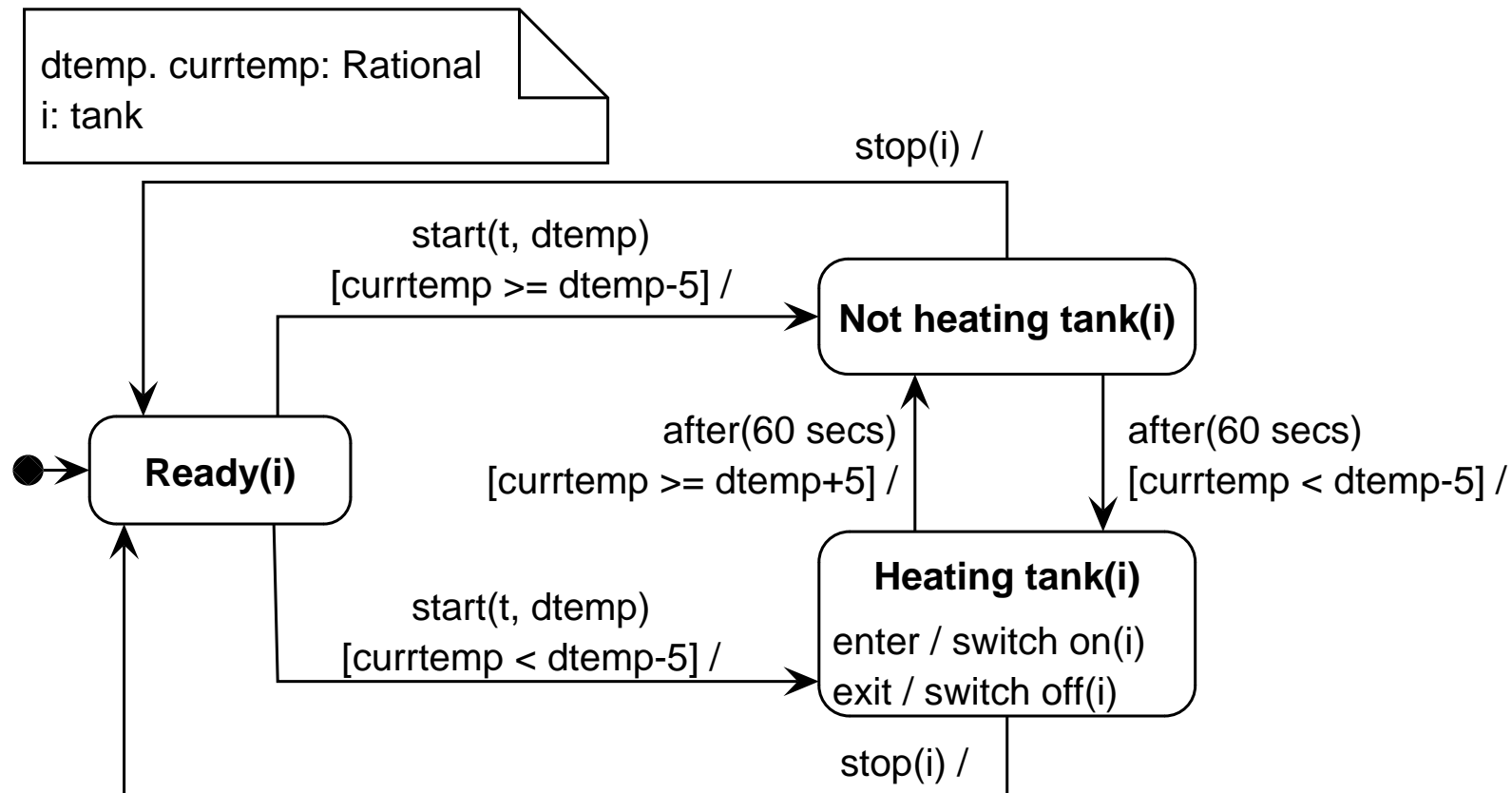
Statestate analogon of DFDs.



Statechart for the control activity



Substatecharts



Temporal events

Timeout:

- For each event e and natural number n , $\text{timeout}(e, n)$, abbreviated to $\text{tm}!(e, n)$, occurs n time units after the most recent occurrence of the event e .

An $\text{after}(n)$ event that leaves state S can be defined as $\text{tm}!(\text{en}(S), n)$.

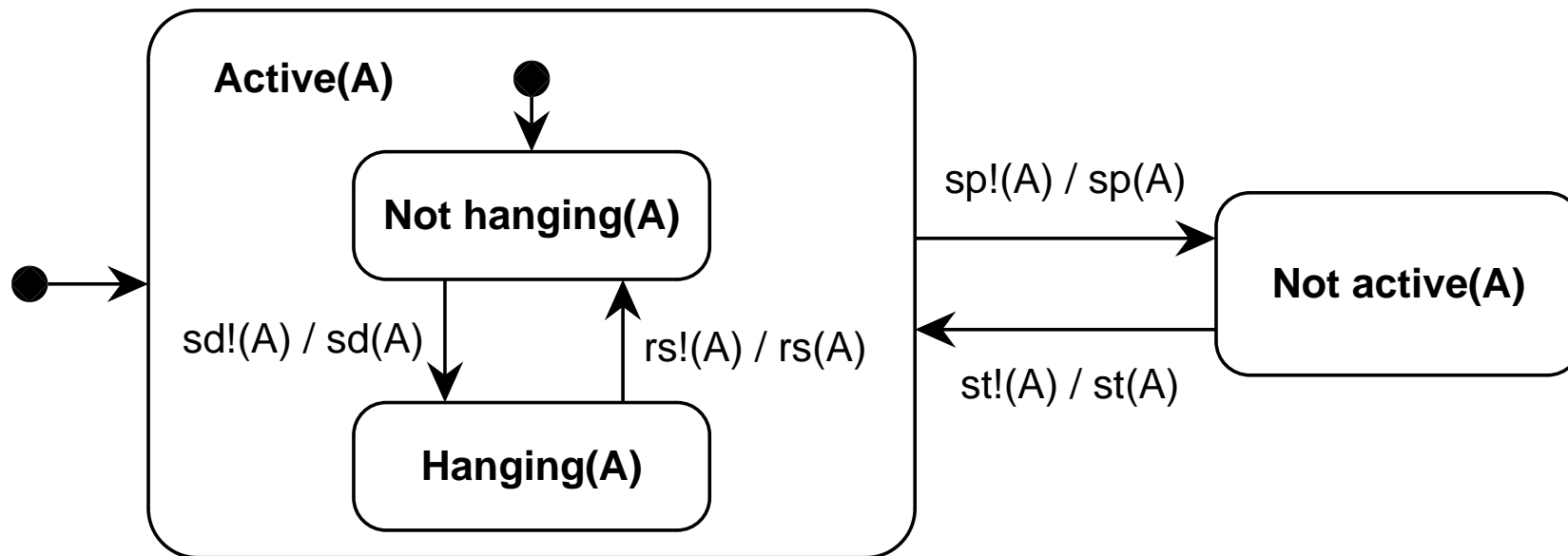
- When a statechart enters a state S , Statemate generates the event $\text{en}(s)$.
- When it exits S , it generates the event $\text{ex}(S)$.

Scheduled action:

- For each action a and natural number n , the action $\text{schedule}(a, n)$, abbreviated $\text{sc}!(a, n)$, schedules the action a to occur exactly n time units later.

$\text{when}(n) / a$ can now be expressed as $\text{sc}!(a, n)$.

Activity states



Changing activity status

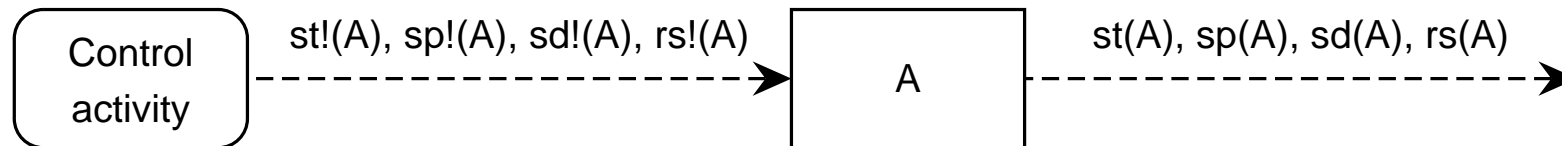
Actions that will cause a change in state of an activity:

sp!(A)	stop(A)
st!(A)	start(A)
sd!(A)	suspend(A)
rs!(A)	resume(A)

Events generated when an activity changes state:

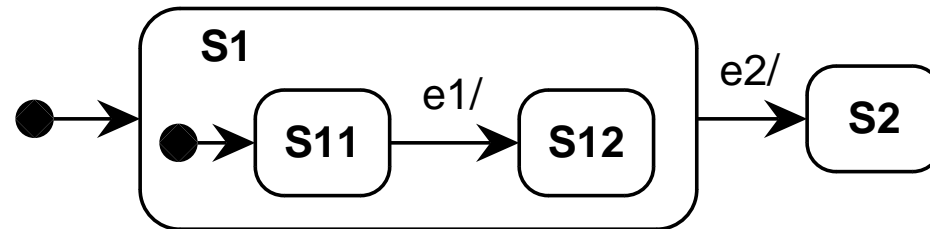
sp(A)	stopped(A)
st(A)	started(A)
sd(A)	suspended(A)
rs(A)	resumed(A)

Interface between control activity and sibling activities

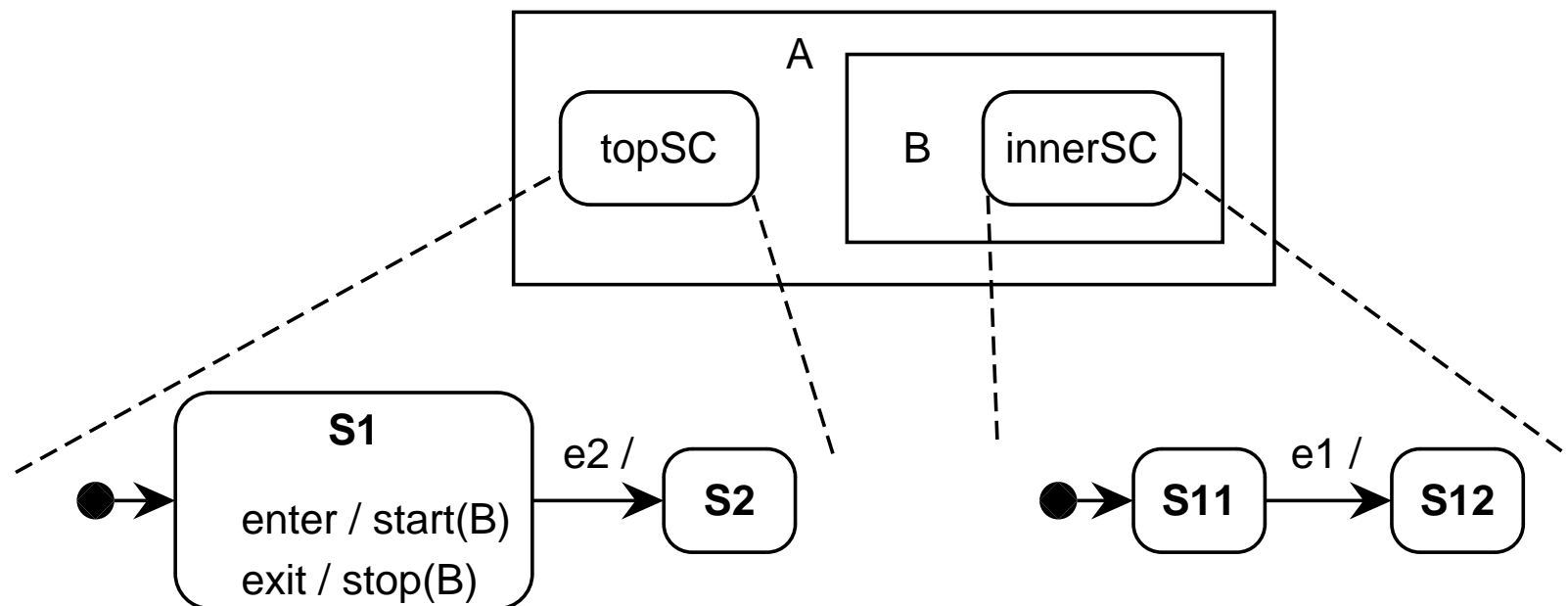


This interface is always present, but it is not shown in an activity chart.

Hierarchy in statecharts and in activity charts (1)

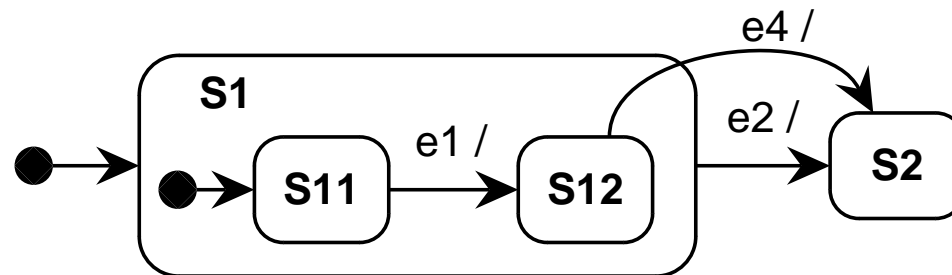


Moving the activity to another statechart leads to an equivalent model (having the same step semantics):

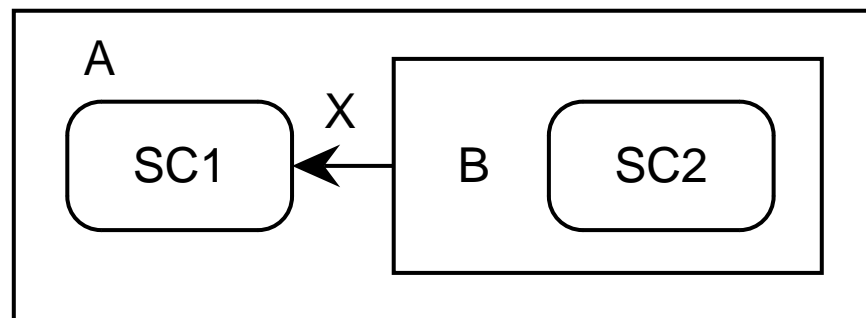


Hierarchy in statecharts and in activity charts (2)

This can be done too if there is an outward transition:

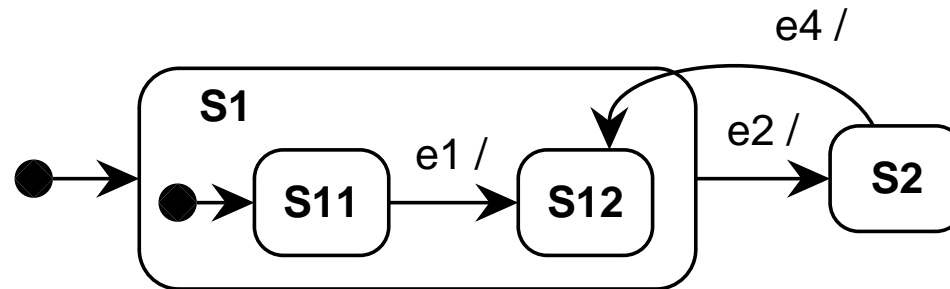


can be replaced by the equivalent model (same step semantics):

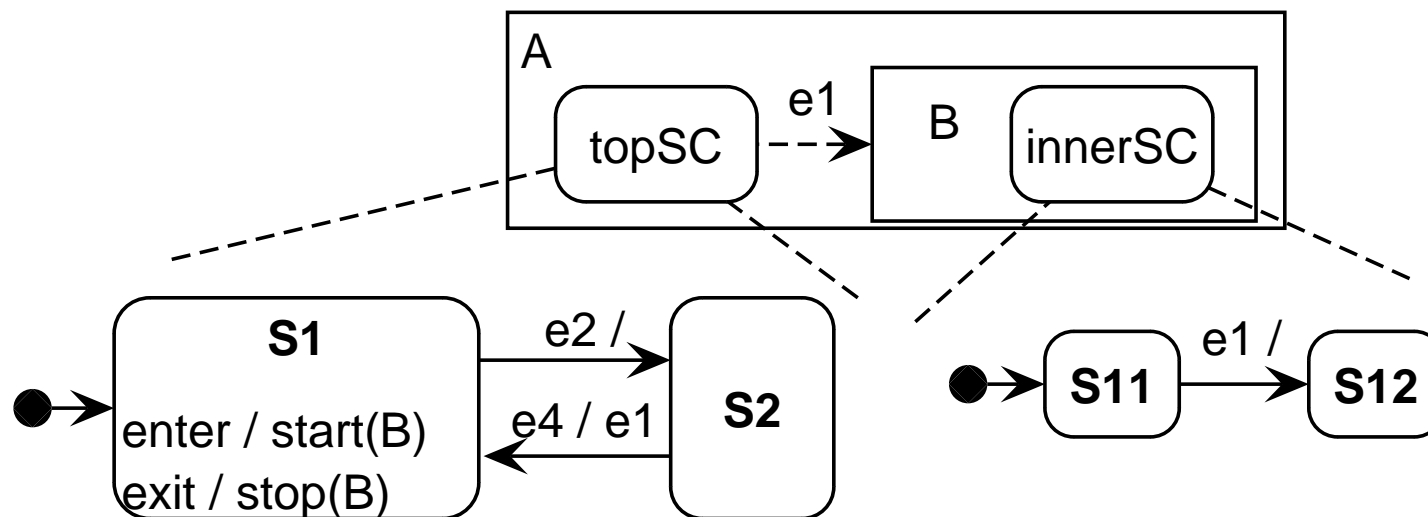


Hierarchy in statecharts and in activity charts (3)

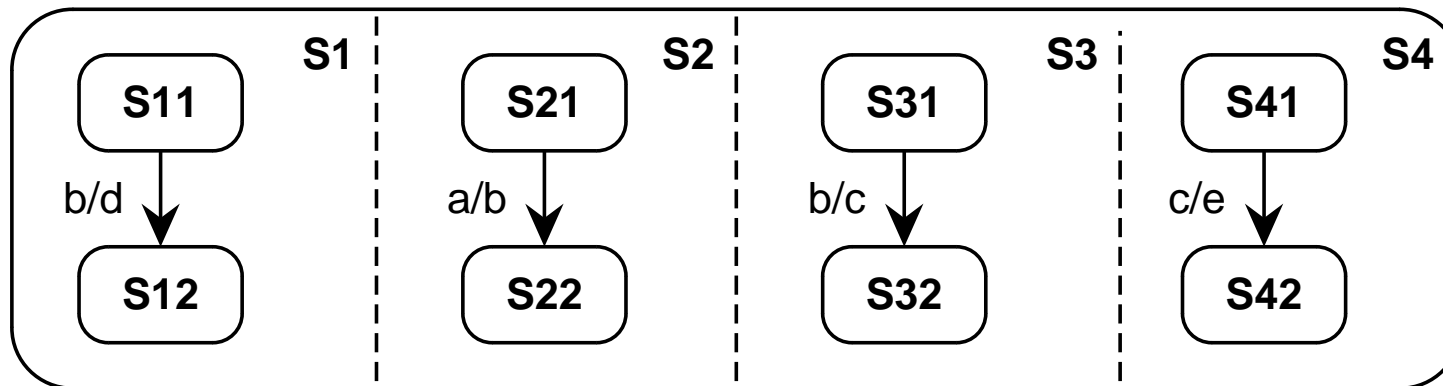
When there is an inward transition this does not work:



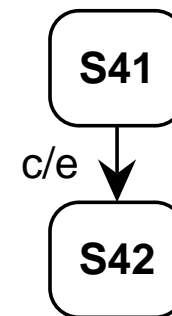
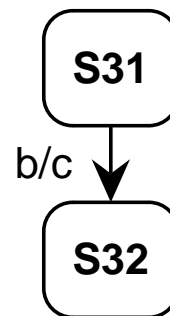
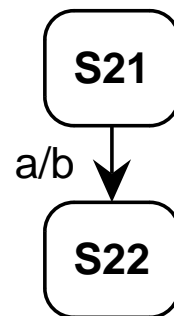
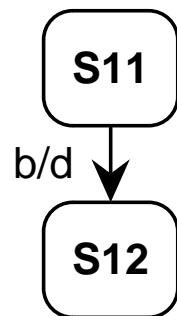
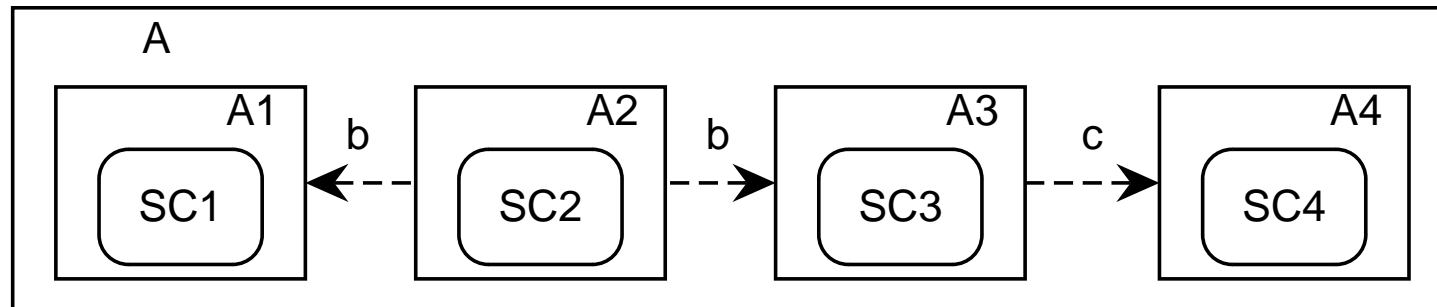
is not equivalent to this model (different step and superstep semantics):



Parallelism statecharts and in activity charts (1)



Parallelism statecharts and in activity charts (2)



Execution semantics (1): Choices made

- *Channel capacity.* Flows have capacity one.
- *Time-continuous flows.* All flows are time-continuous. They behave as data stores.
- *Input buffer.* Each activity has an input buffer, which is a set.
- *Priority.* If a state transition conflicts with a state reaction, then the transition has priority. If two transitions are in conflict, then the highest-level one has priority.
- *Step semantics.* In a step, the system responds to all events that occurred since the start of the previous step.
- *Perfect technology.* Steps do not take time.
- *Breadth-first.* Activities are executed breadth-first.

Execution semantics (2): Execution state

- The status of the system, which consists of the following items:
 - The current configuration of the statecharts;
 - The values of all variables;
 - The truth-values of all conditions;
 - The activation state of each activity;
 - A list of internal events generated during the previous step;
 - A list of outstanding timeout events;
 - A list of outstanding scheduled actions.
- The time currently indicated by the system clock;
- A list of external events that have occurred since the beginning of the previous step.

Execution semantics (3): Execution step

Time is T during the step.

1. Construct the set E of events to be responded to. First, set $E := \emptyset$.
 - (a) Collect all external events generated by the environment since the previous step and add them to E .
 - (b) Collect all events generated in the previous step and add them to E .
 - (c) Collect all events generated by the events in E , and add them to E (recursively).
 - (d) Execute all scheduled actions whose time is due in $(T, T+1]$.
 - (e) Process the list of timeout events. For each $tm(e, n)$ in the list,
 - if $e \in E$ compute and record the time at which $tm(e, n)$ is to be generated;
 - else if the time for $tm(e, n)$ to occur falls in the interval $(T, T+1]$, then generate the event e and deactivate the timeout event.

2. Construct from E a maximal step S to be executed. First, set $S := 0$. Then:
 - (a) Compute the set En of enabled transitions and state reactions, i.e. those whose triggering event occurred and whose guard is true. Remove from En those transitions or reactions that are in conflict with a transition of higher priority in En .
 - (b) Compute from En maximal nonconflicting sets S of transitions. Add to each set the state reactions that do not conflict with it. Each resulting set is called a *step*.
 - (c) Select a step S to be executed.
3. Execute S .
 - (a) Add scheduled actions in the step to the list of scheduled actions.
 - (b) Perform all other actions in the step.
 - (c) Update the information on the history of states.
 - (d) Update the current configuration.

Execution semantics (4): Superstep execution

1. Construct the set E of events to be responded to as for a step.
2. Repeat the following until $E = \emptyset$:
 - 2.1 Construct from E a maximal system step S to be executed as for a step. (There are no internal events at the beginning of a superstep.)
 - 2.2 Execute S as in the step execution semantics.
 - 2.3 Reconstruct the set E of events to be responded to. First, set $E := \emptyset$. Then:
 - (a) Collect all events generated on internal flows and add them to E .
 - (b) Generate derived events from the events in E and add these to E (recursively).

Main points

- Activity charts are a syntactic variant of DFDs.
- Statecharts have an execution semantics in STM.
- The presence of hierarchy and parallelism in statecharts as well as in activity charts increases the number of specification options for the author of a specification.

Chapter 22. The Unified Modeling Language (UML)

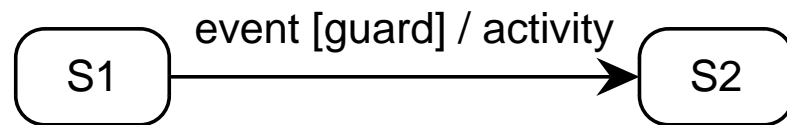
- Attempted unification of notations for object-oriented software design.
- Industrial standard defined by the OMG.
- Standard is still being updated.
- Different books present different versions!
- We treat only a light-weight version, expected to be consistent with future versions.

The UML contains eight notations

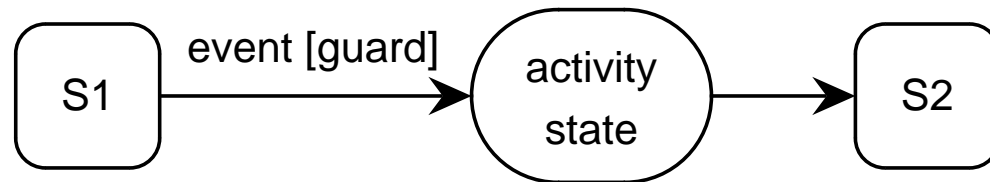
UML notations can be used in different methods in different ways. We will use them as follows. (Notations between brackets are not used by us.)

- **Activity diagrams.** User workflow.
- (Use case diagrams. System functions and context.)
- **Static structure diagrams.** Decomposition into software objects.
- **Statecharts.** Object life cycles.
- **Sequence diagrams.** Message-passing during a scenario.
- **Collaboration diagrams.** Message-passing during a scenario.
- (Component diagrams. Dependencies between executables.)
- (Deployment diagrams. Network of computing resources.)

Activity diagrams



- A transition in a statechart is instantaneous.
- If we want to represent that an action takes time, turn it into a **activity state**:



- Transitions are still instantaneous.

Workflow

- We will use activity diagrams to represent user workflows.
- Each individual workflow handles a case.
- Each individual workflow has local variables: state of the workflow, state of the case, list of events to be responded to.
- The variables are usually stored in a database.
- Guards are conditions on these variables.

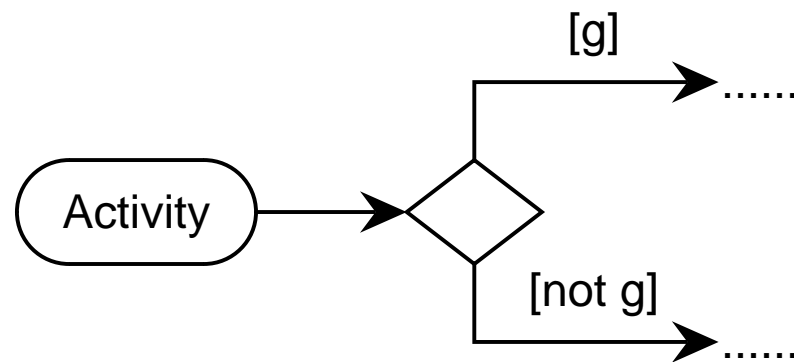
Representing sequence



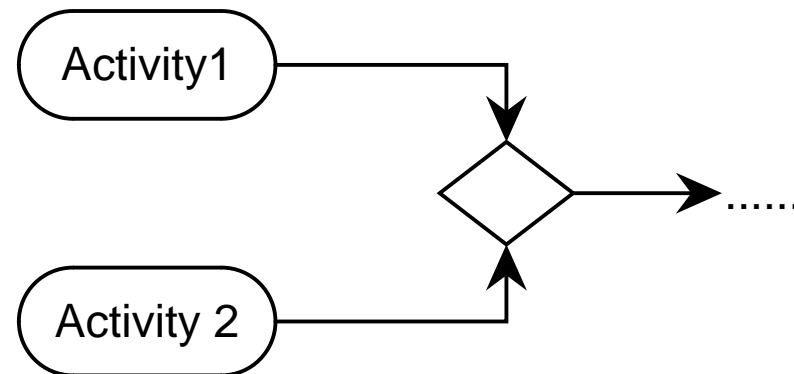
(k) When activity 1 terminates, activity 2 starts.

(l) When activity 1 terminates, the workflow waits for *e* to occur when *g* is true, before starting activity 2.

Representing choice



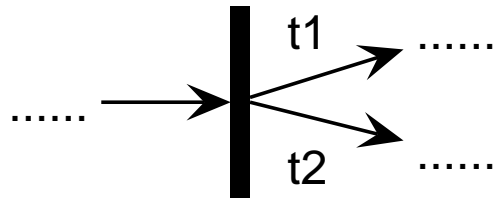
(m) Or-split.



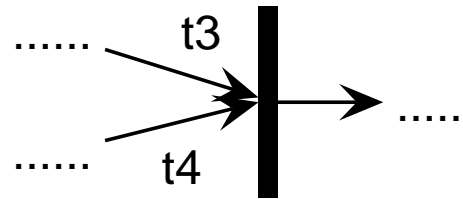
(n) Or-join.

Guards are tested on the workflow variables.

Representing parallelism

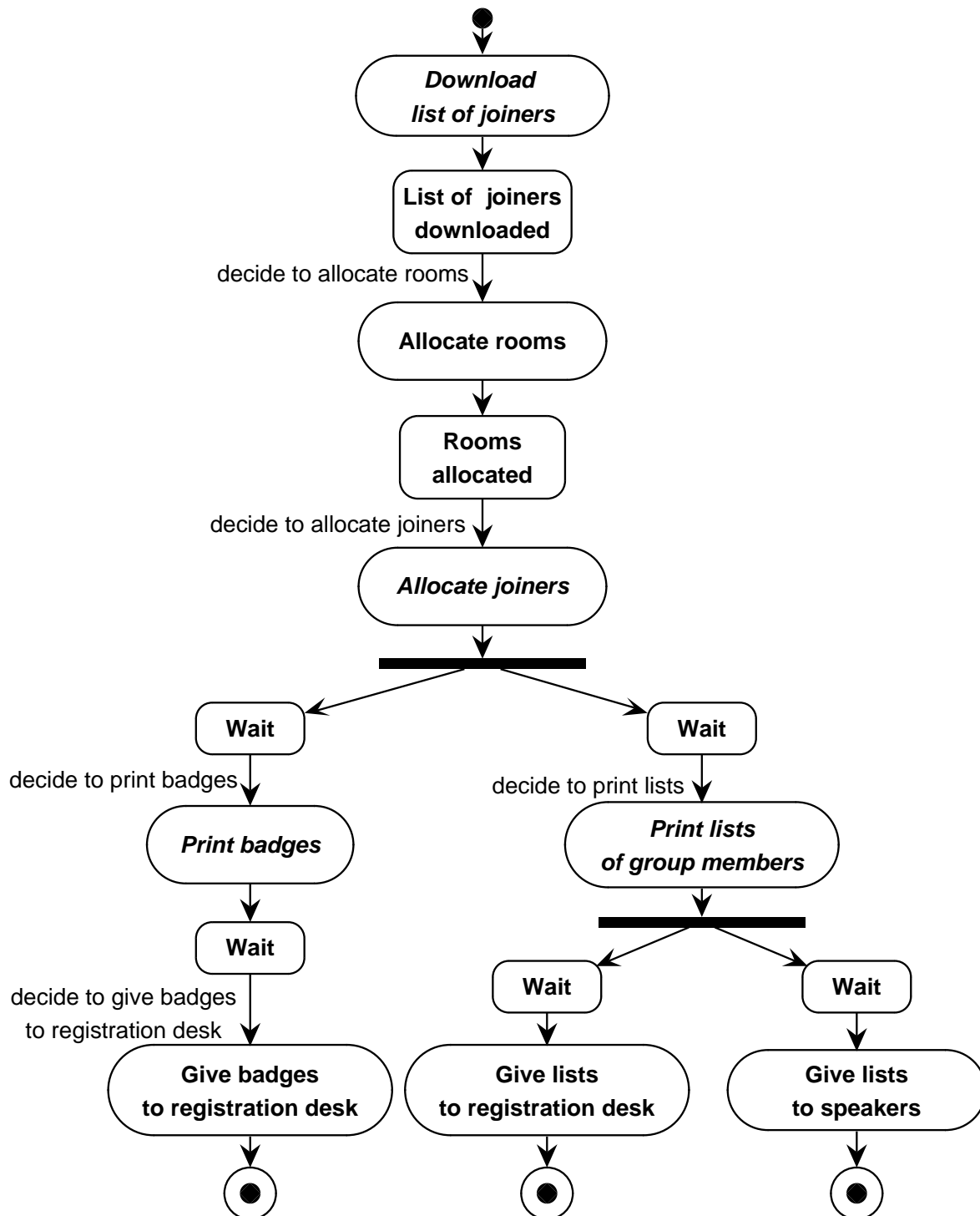


(o) And-split. t1 and t2 are executed simultaneously.



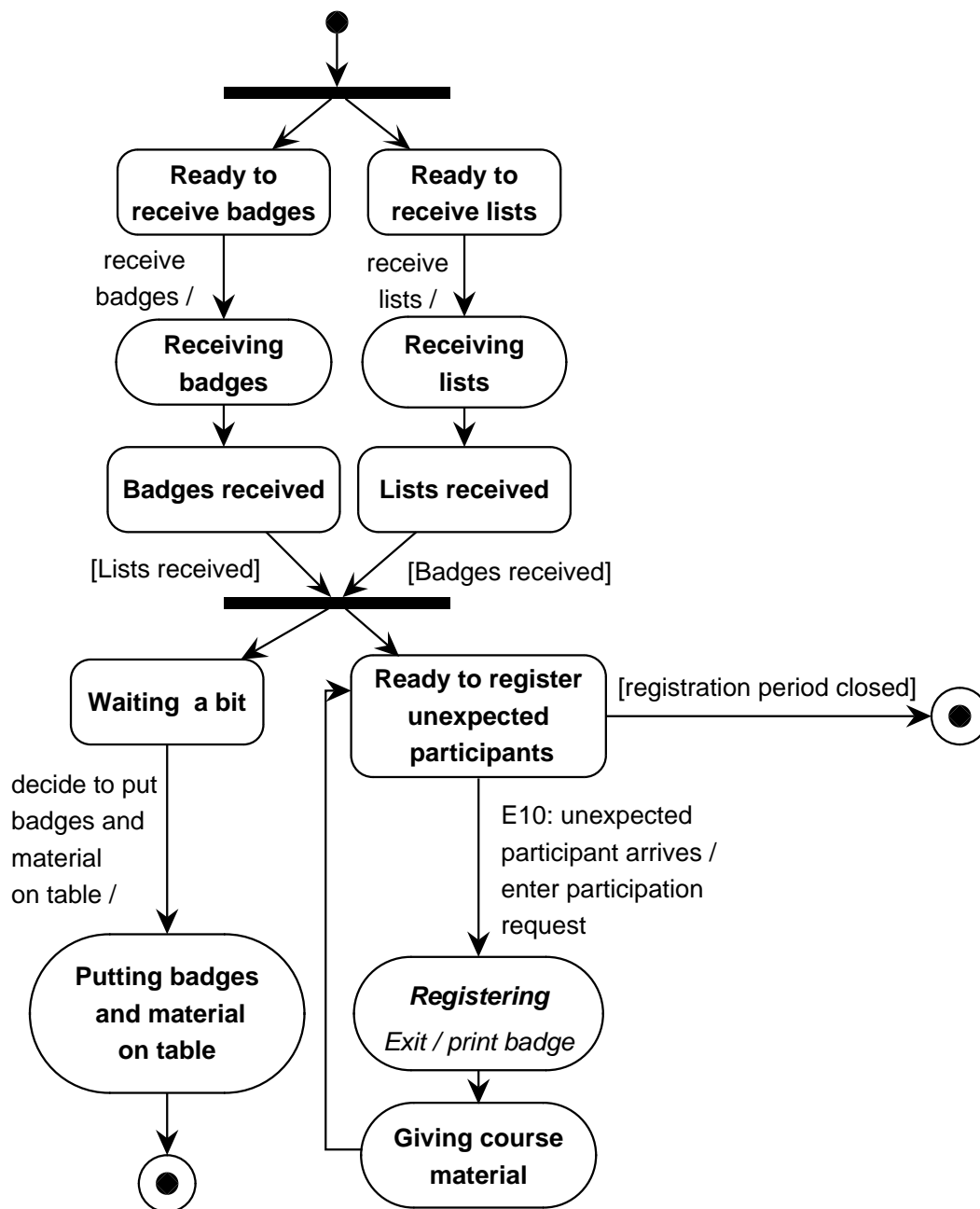
(p) And-join. t3 and t4 are executed simultaneously.

Workflow of the course coordinator



Activities in italics to be supported by TIS.

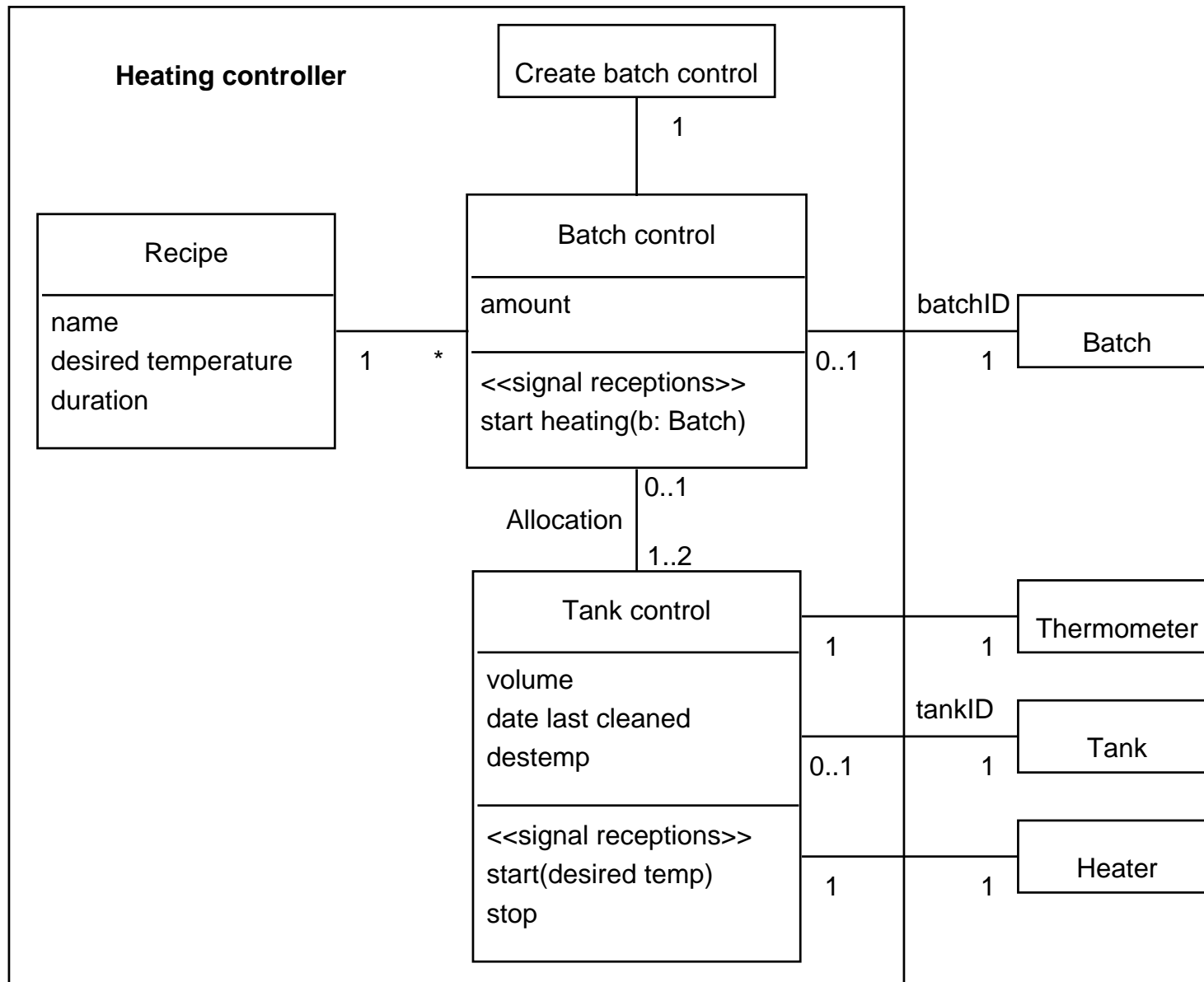
Workflow at the registration desk



More examples in appendix H
(www.mkp.com/dmrs)

Static Structure Diagrams (SSDs)

- Also known as *class diagrams*.
- An **object** is a software entity with a fixed identity, a local state and an interface through which it offers services to its environment.
- An **object class** is an object type.
 - Class intension: All properties shared by all instances.
 - Extension: All possible instances.
 - Extent: Set of currently existing instances.



Meaning of an SSD

- Which classes of objects can exist in the software system,
- and how many of them can exist.
- Attributes are local variables of the object.
- Associations are access paths!
- Services can be operations or signal receptions.
 - **Operation** = computation performed by object.
 - **Signal** = named data structure that can be sent as a message to objects.

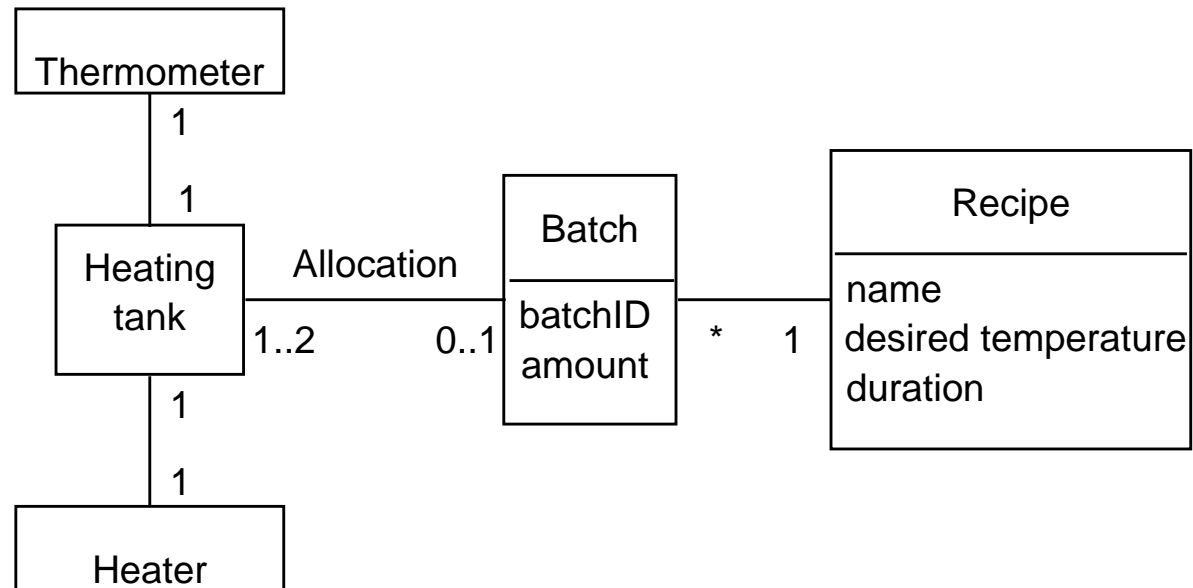
ERDs and SSDs

ERD	SSD
entity type	class
entity	object
relationship	association
tuple	link
association entity	association object
association entity type	association class
cardinality property	multiplicity property

The major differences are:

- ERD used to represent structure of subject domain.
- SSD used to represent structure of software objects.
- Objects offer services.

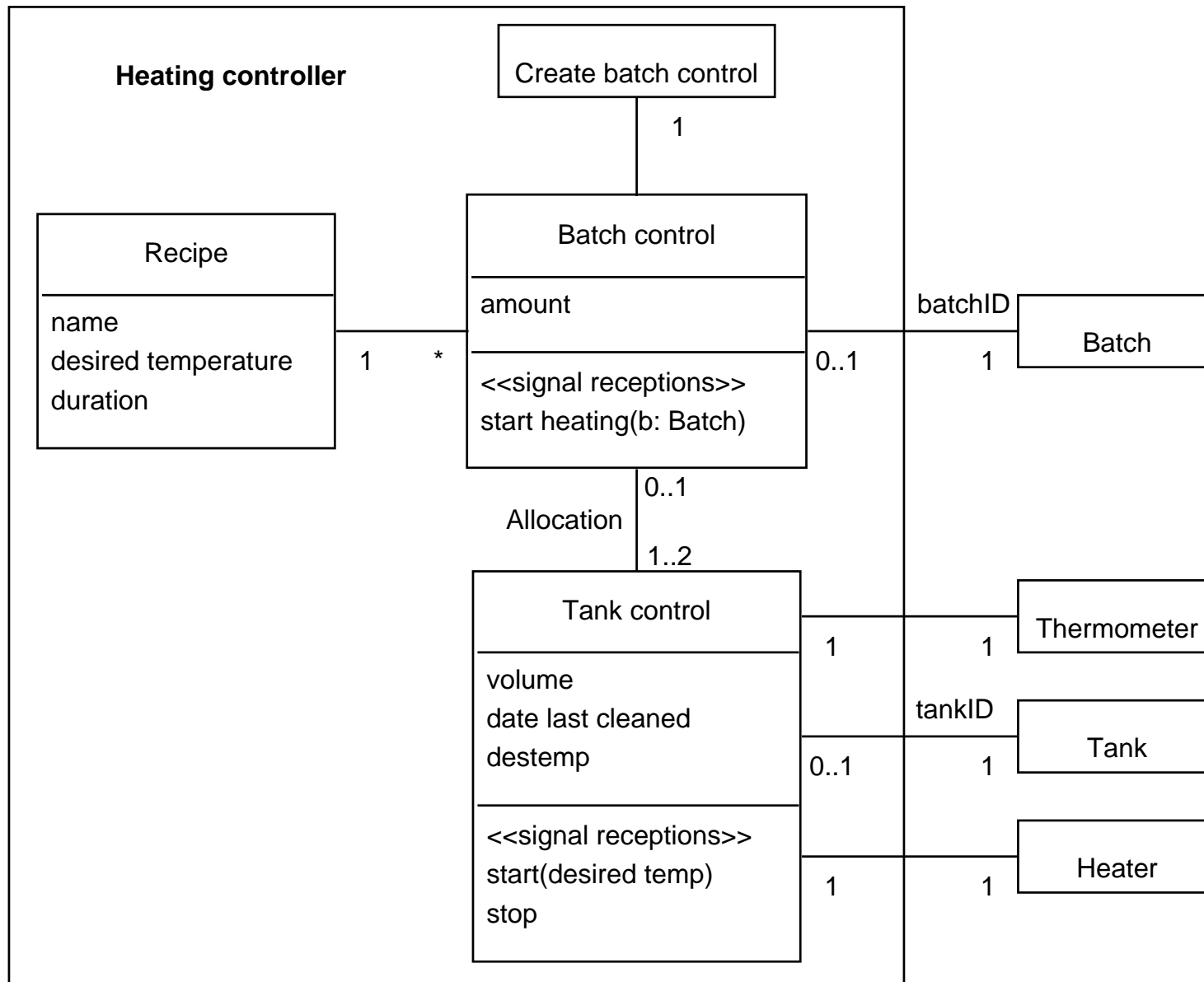
Subject domain of the heating controller



Guidelines used to design heating controller:

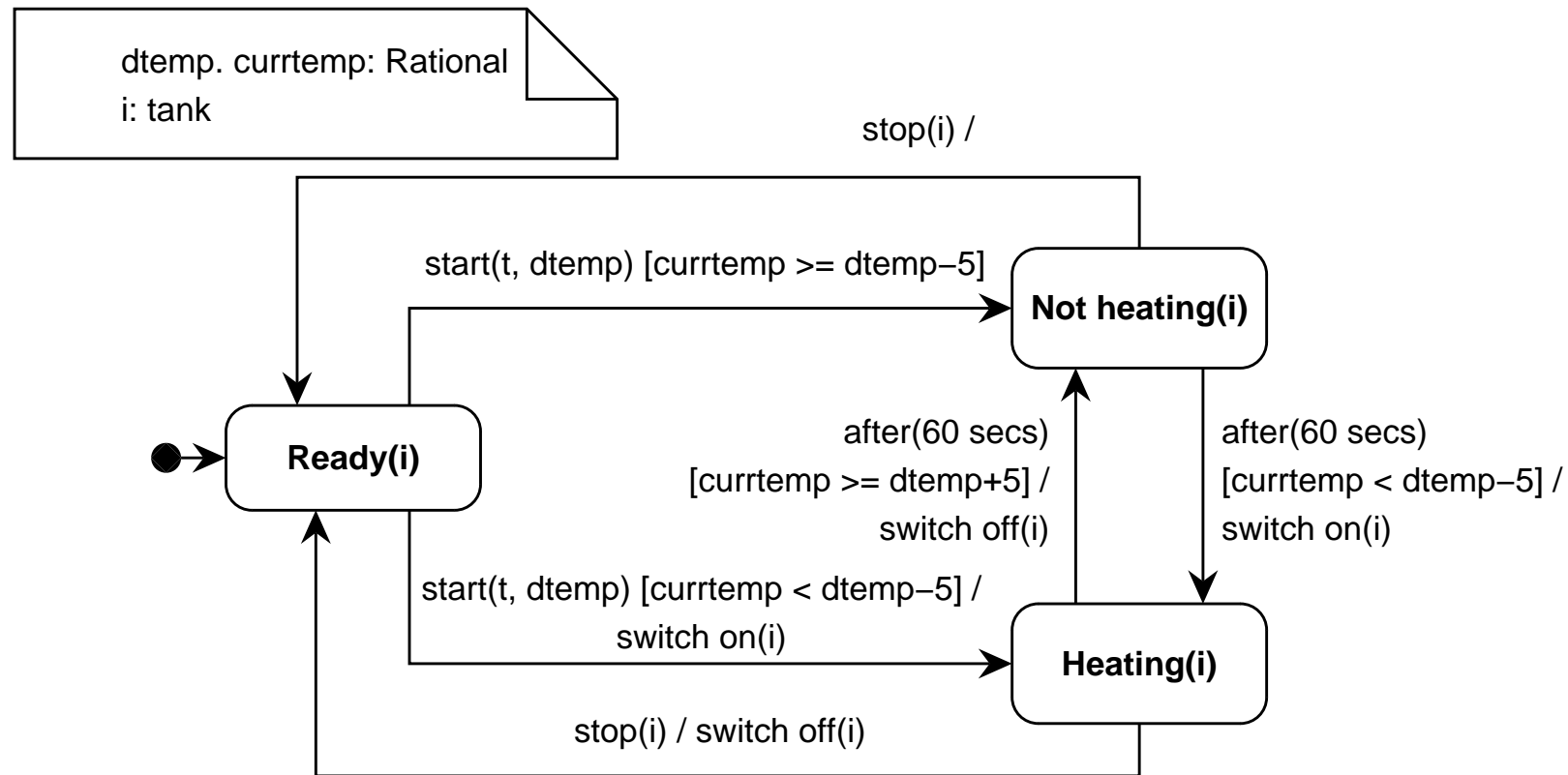
- Subject-orientation
- Functional decomposition
- Device-orientation

How are these guidelines used?



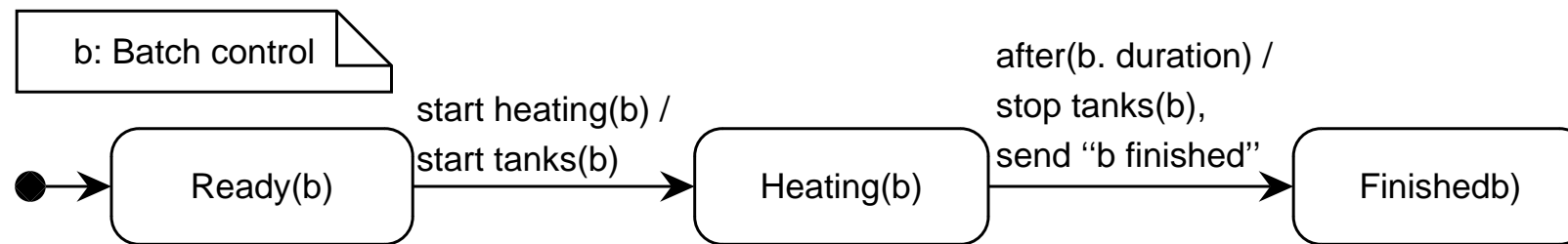
Statecharts in the UML (1)

Tank control



Statecharts in the UML (1)

Batch control

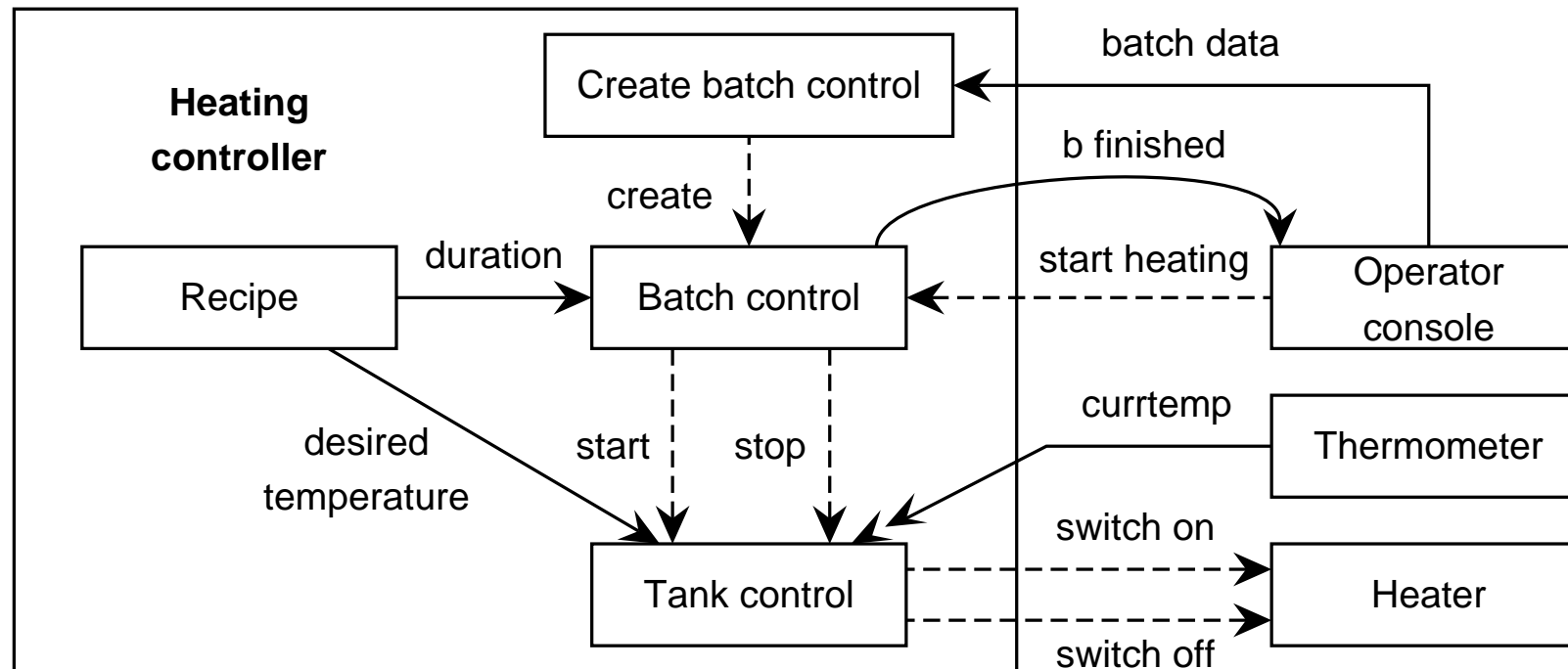


Dictionary:

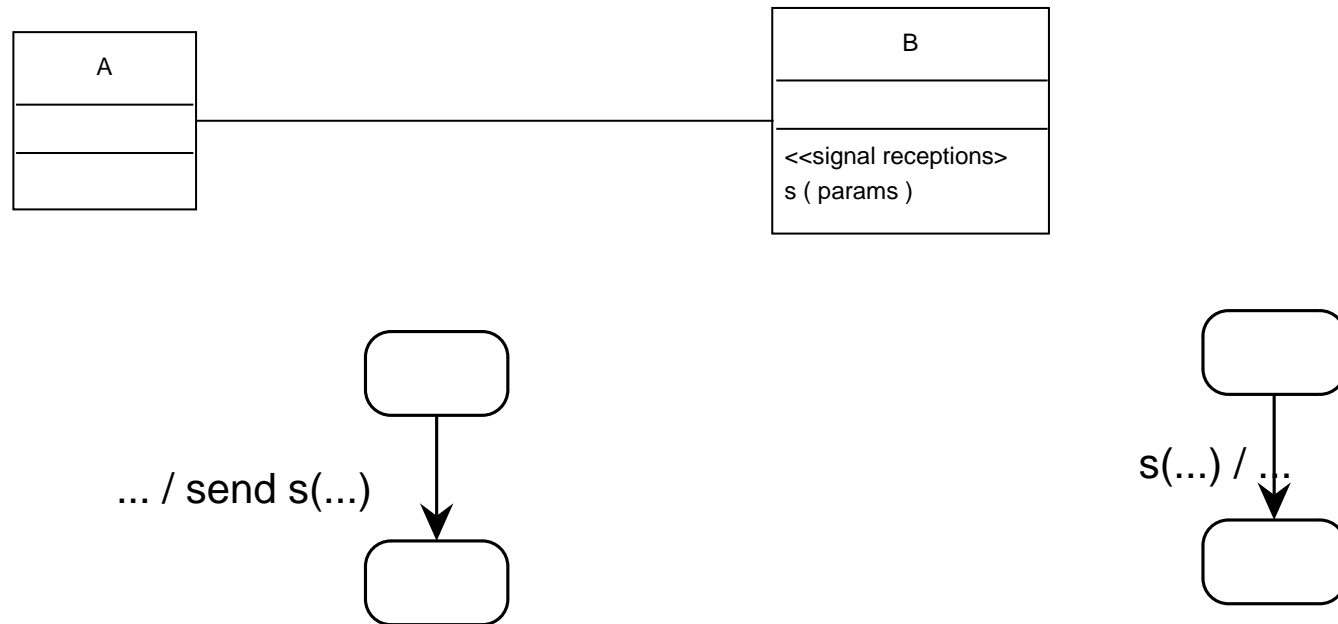
- **start_tanks(b: Batch).**
 - For all tanks `t` in `b.heating_tank`, send `start(t, b.recipe.desired_temperature)`.
- **stop_tanks(b: Batch).**
 - For all tanks `t` in `b.heating_tank`, send `stop(t)`.

Coherence between SSD and statecharts (1)

- Attributes, identifiers, interface data declared as local variables.
- Signals can trigger transitions.
- Actions can be operation calls or signal sending.
- Illustrate this coherence by an architecture diagram:



Coherence between SSD and statecharts (2)

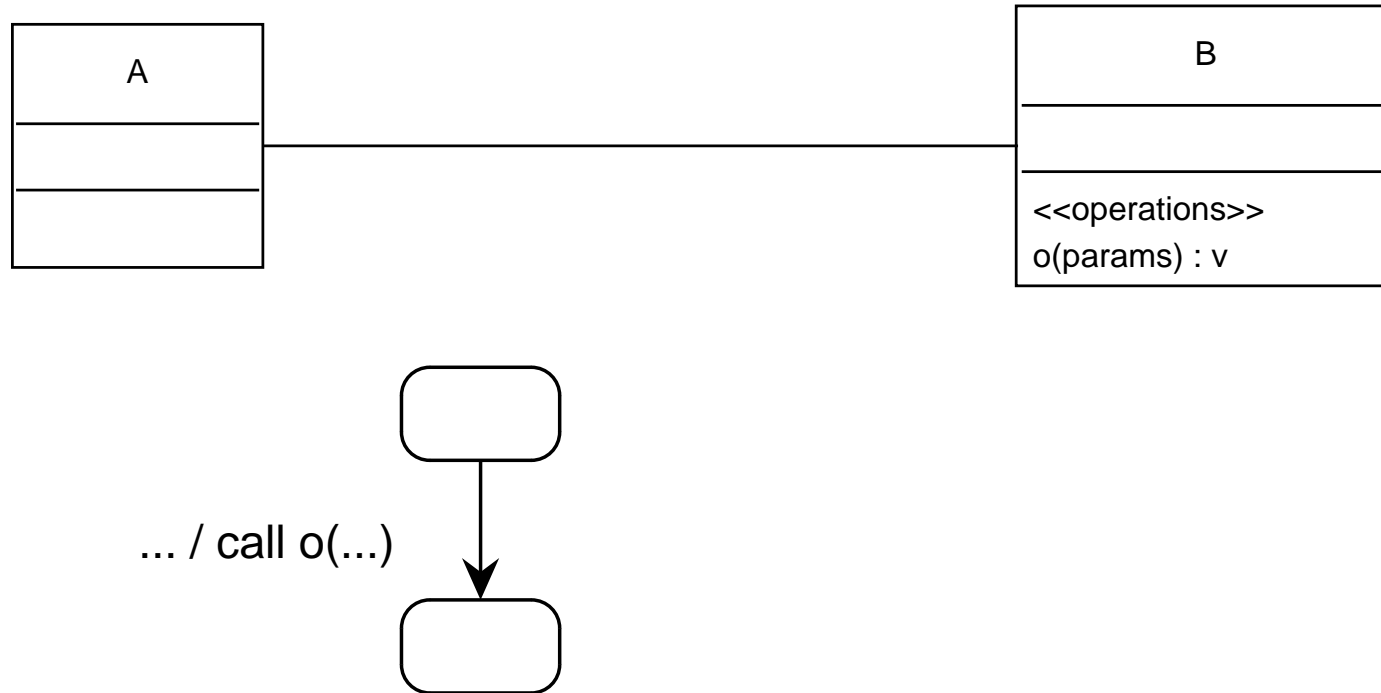


Statechart of instances of A.

Statechart of instances of B.
Transitions can be triggered by s(...).

Instances of A can send messages to instances of B because there is an association from A to B.

Coherence between SSD and statecharts (3)



Statechart of instances of A.

Operation o must be defined for instances of B.

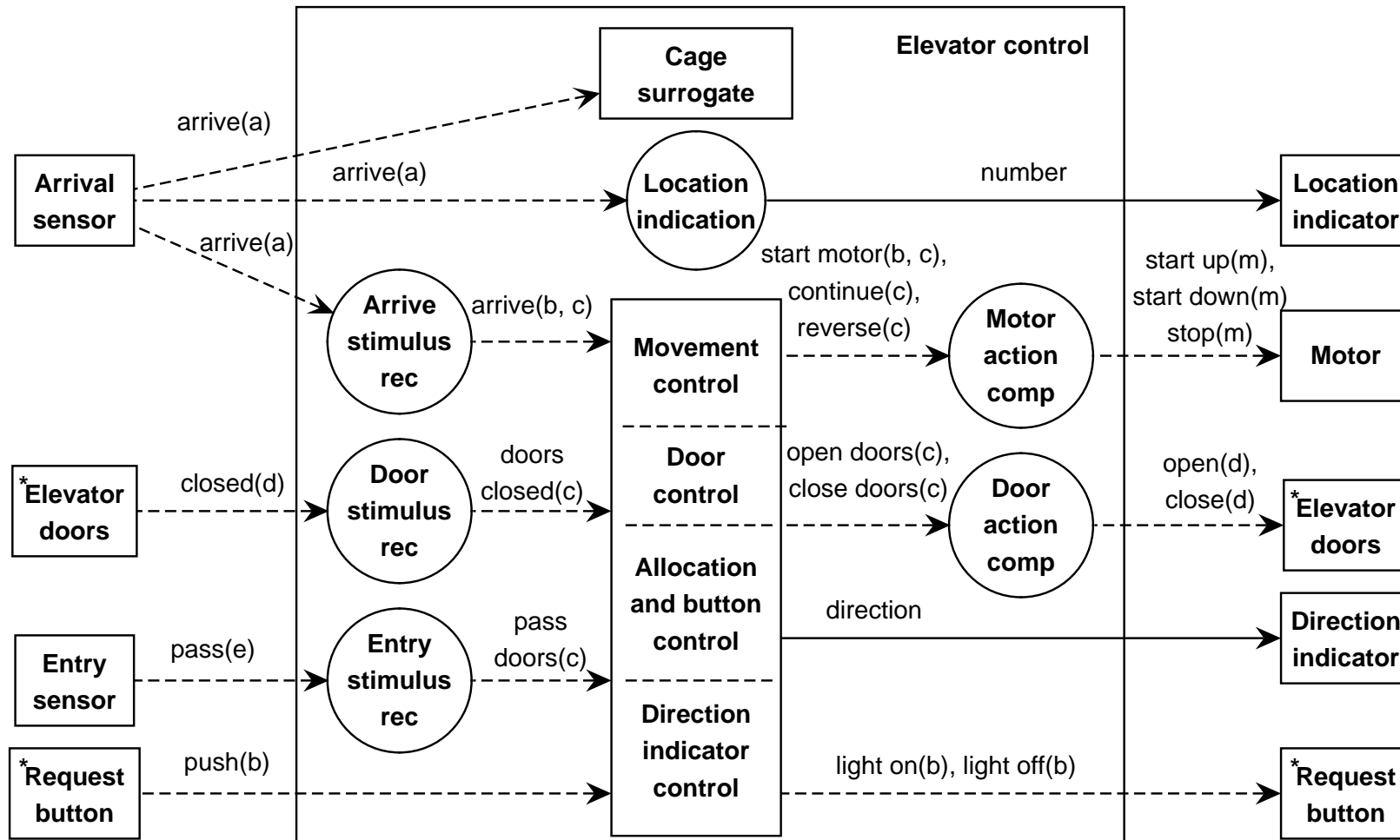
How to find an SSD

Considerations:

- A communication diagram shows communication channels among components.
- An SSD shows cardinality properties of components and access paths between classes.
- The SSD shows the access paths needed by each object to do its job.
- To find the access paths, we need to find out the job of each object first.
- To find the job of each object, we draw the communication diagram first.

Guidelines for finding an SSD

- ✓ Consider one stimulus-response pair.
- ✓ Draw a communication diagram of the stimulus-response process.
 - Use architecture guidelines to find components. See section 19.4.
- ✓ Draw an SSD of the classes and access paths needed by each component in the communication diagram.
 - Definition of operations tell us where each object must find its data.
 - Use subject domain ERD as inspiration to find multiplicity properties.

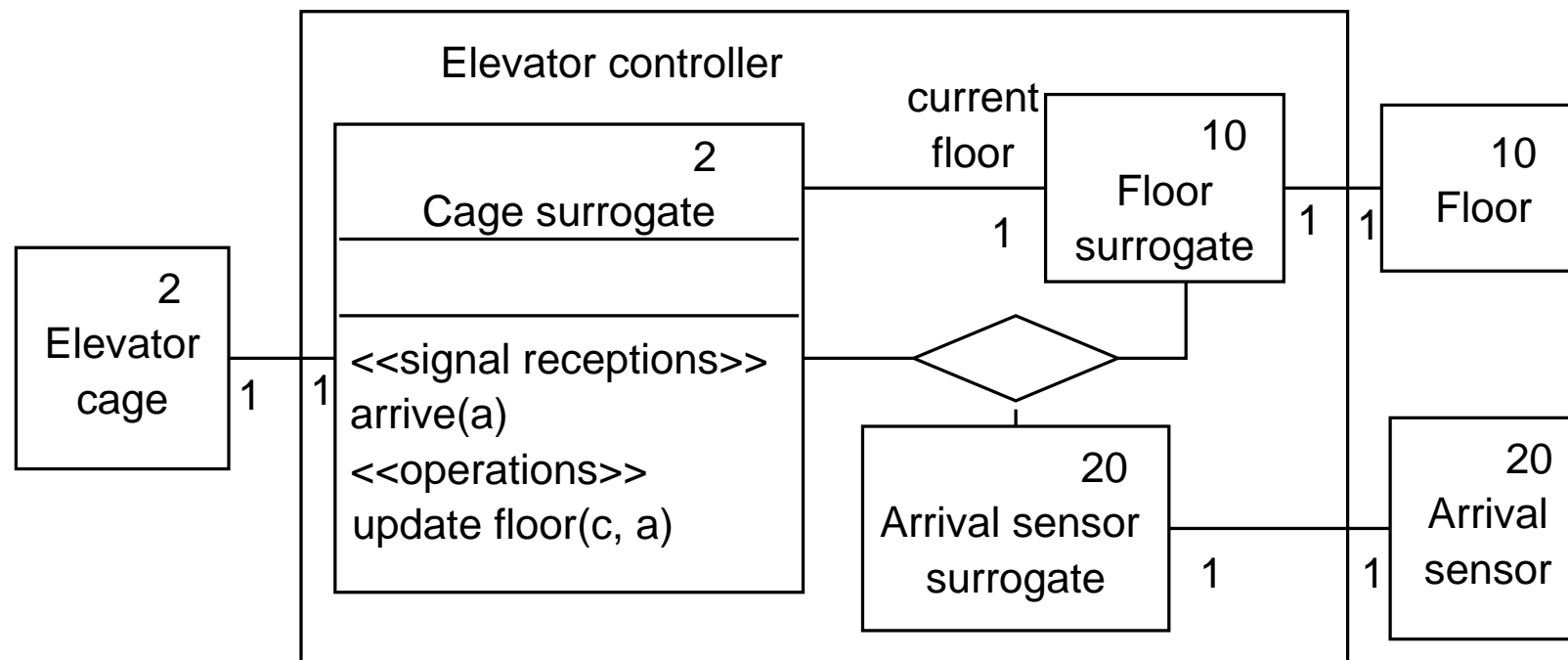


What information does a cage surrogate need? Here is its job:

c: Cage surrogate

arrive(a) / update_floor(c, a), where $\text{update_floor}(c, a) = c.\text{current_floor} := a.\text{floor}$.

Here are the access paths needed:

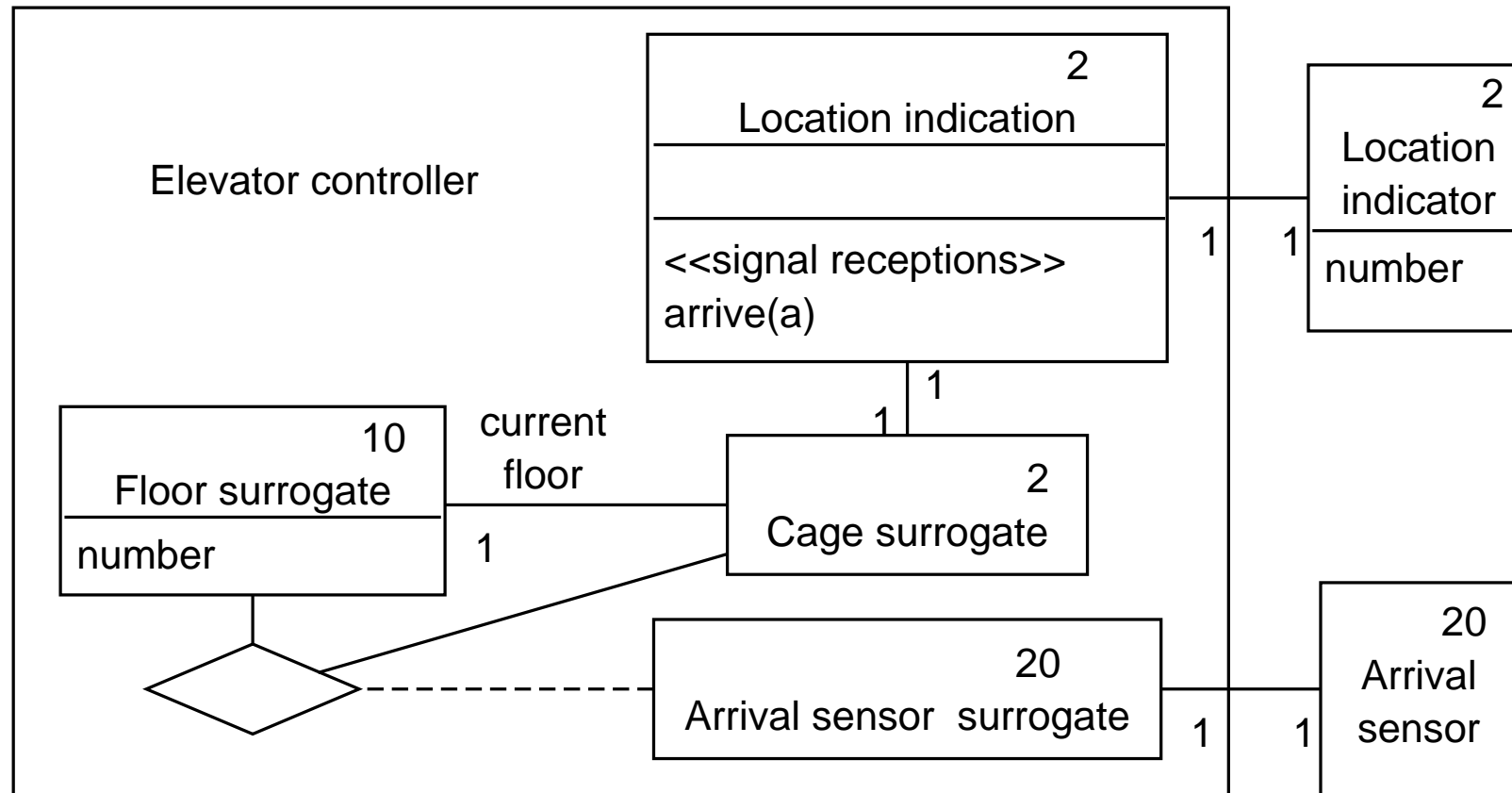


Here is the job of the location indicator:

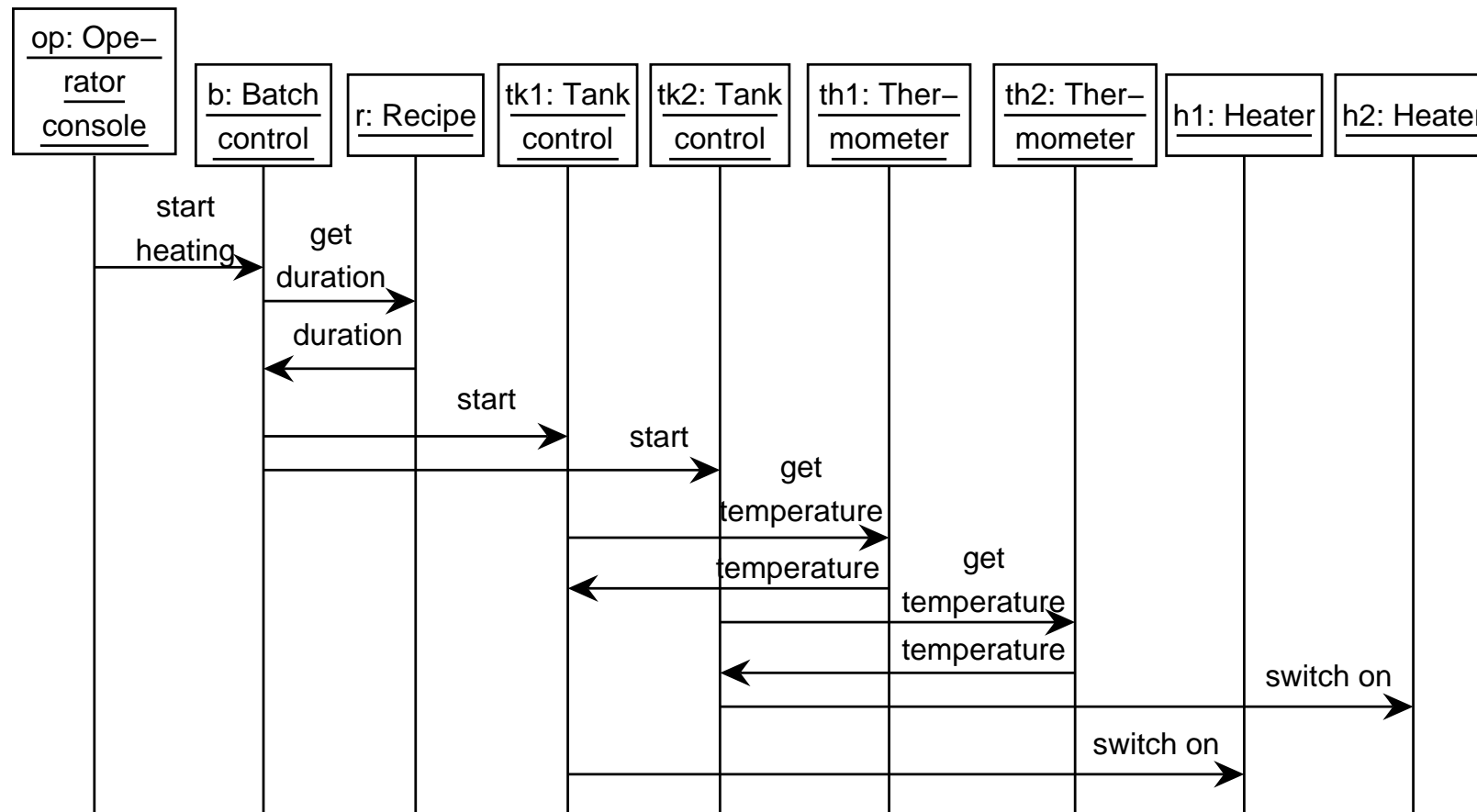
Location indication:

- When arrive(a),
- do set number(a.cage.location_indicator, a.cage.current_floor.number).

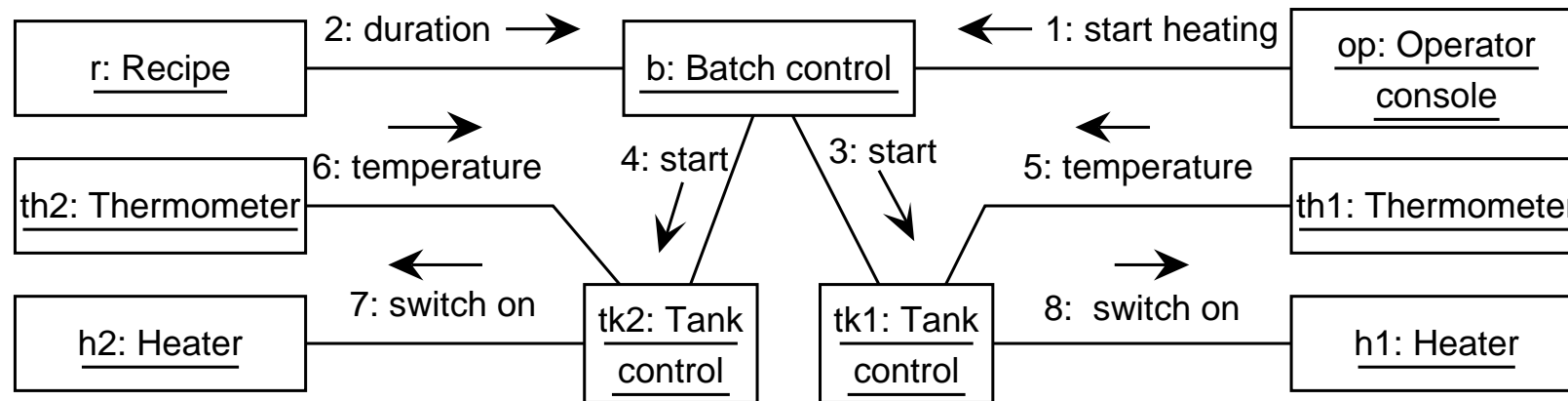
Here are the access paths through which it gets its information:



Sequence diagrams can be used to illustrate scenarios



Collaboration diagrams can be used to illustrate scenarios too



Possible uses of sequence and collaboration diagrams

- As illustration: The system must be able to execute this scenario.
- As specification: All executions of the system must contain this scenario.
- To specify patterns: The diagram represents roles that objects in the system play.

Main points

- Activity diagrams can be used to specify user workflow.
- Class diagrams represent decomposition of system into objects.
- Statecharts can be used to represent object life cycles.
- Use architecture diagrams to keep track of relationship between SSDs and statecharts.
- Find SSD by identifying access paths needed in stimulus-response processing.
- Illustrate executions by means of sequence or collaboration diagrams.

Chapter 23. Not Yet Another Method

The weight of professional boxers is classified according to the following scheme:

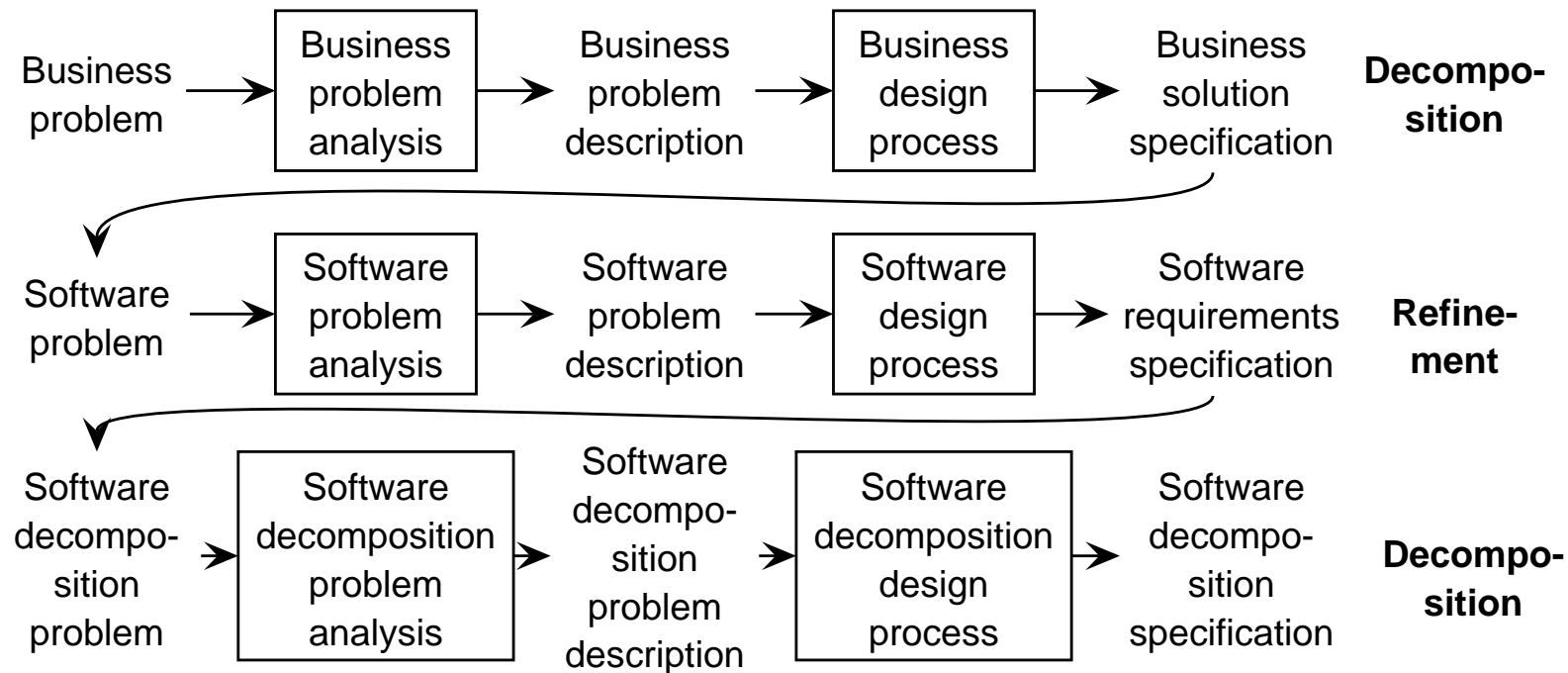
flyweight	≤ 112 pounds
bantamweight	≤ 118 pounds
featherweight	≤ 126 pounds
lightweight	≤ 135 pounds
welterweight	≤ 147 pounds
middleweight	≤ 160 pounds
heavyweight	> 160 pounds

We can learn two things from this classification:

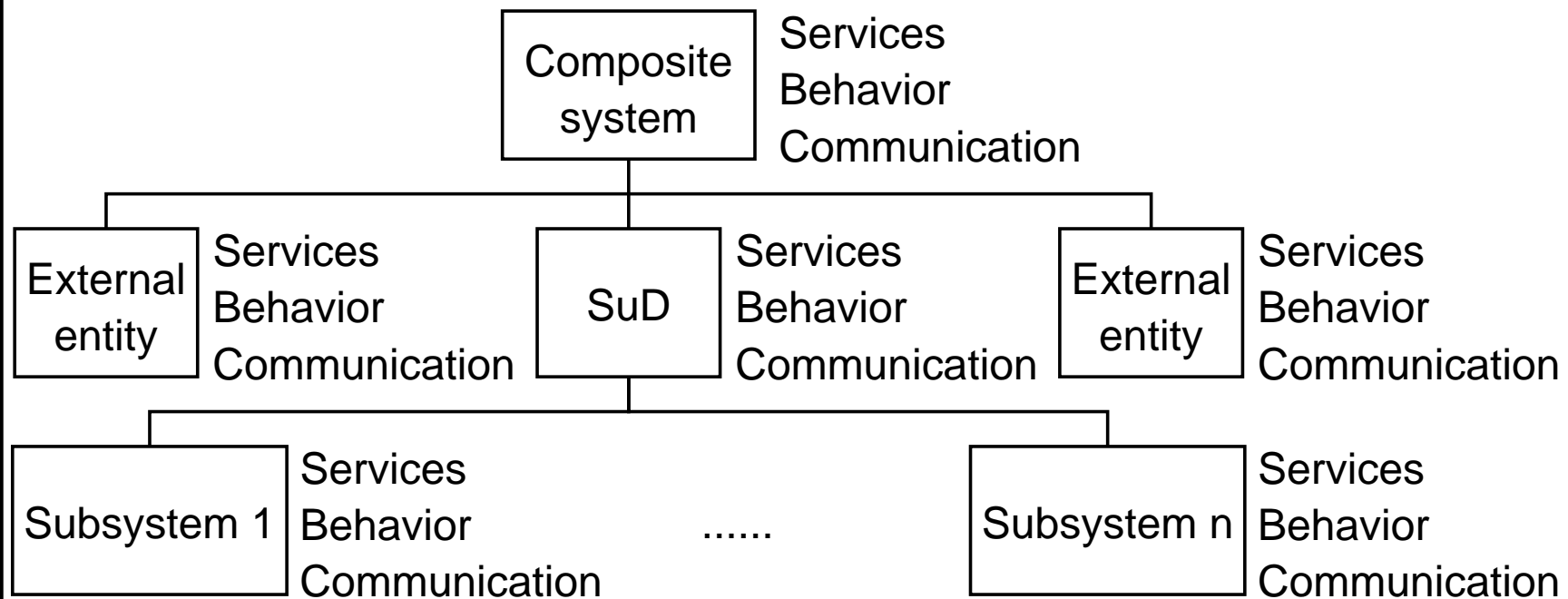
1. Lightweight is not the lightest weight.
2. Heavyweights can be as heavy as they want.

Keep it simple.

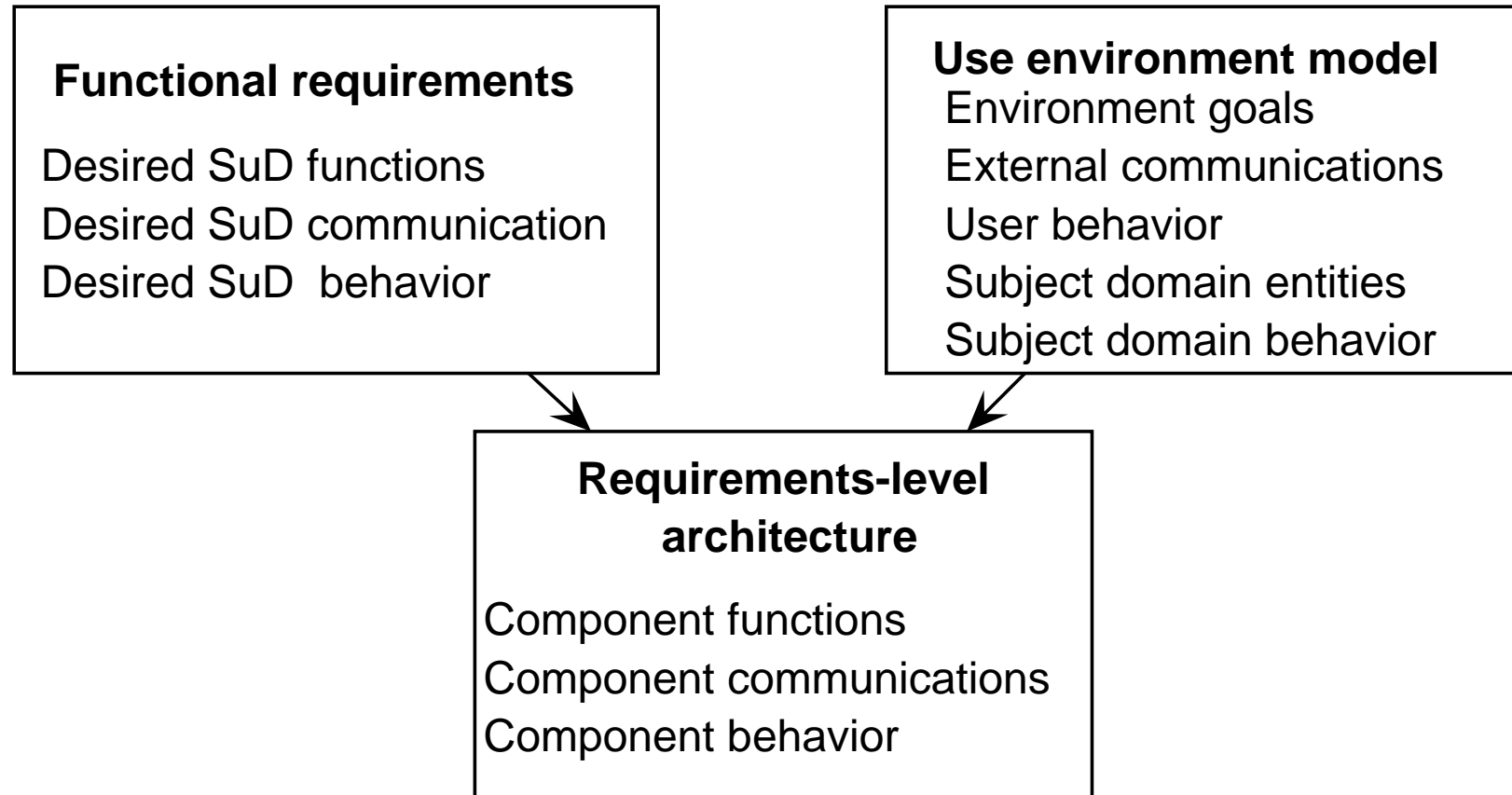
Problem-solving levels



The system hierarchy



Software design approach



- Requirements-level decomposition.
- Independent from changes in implementation platform.

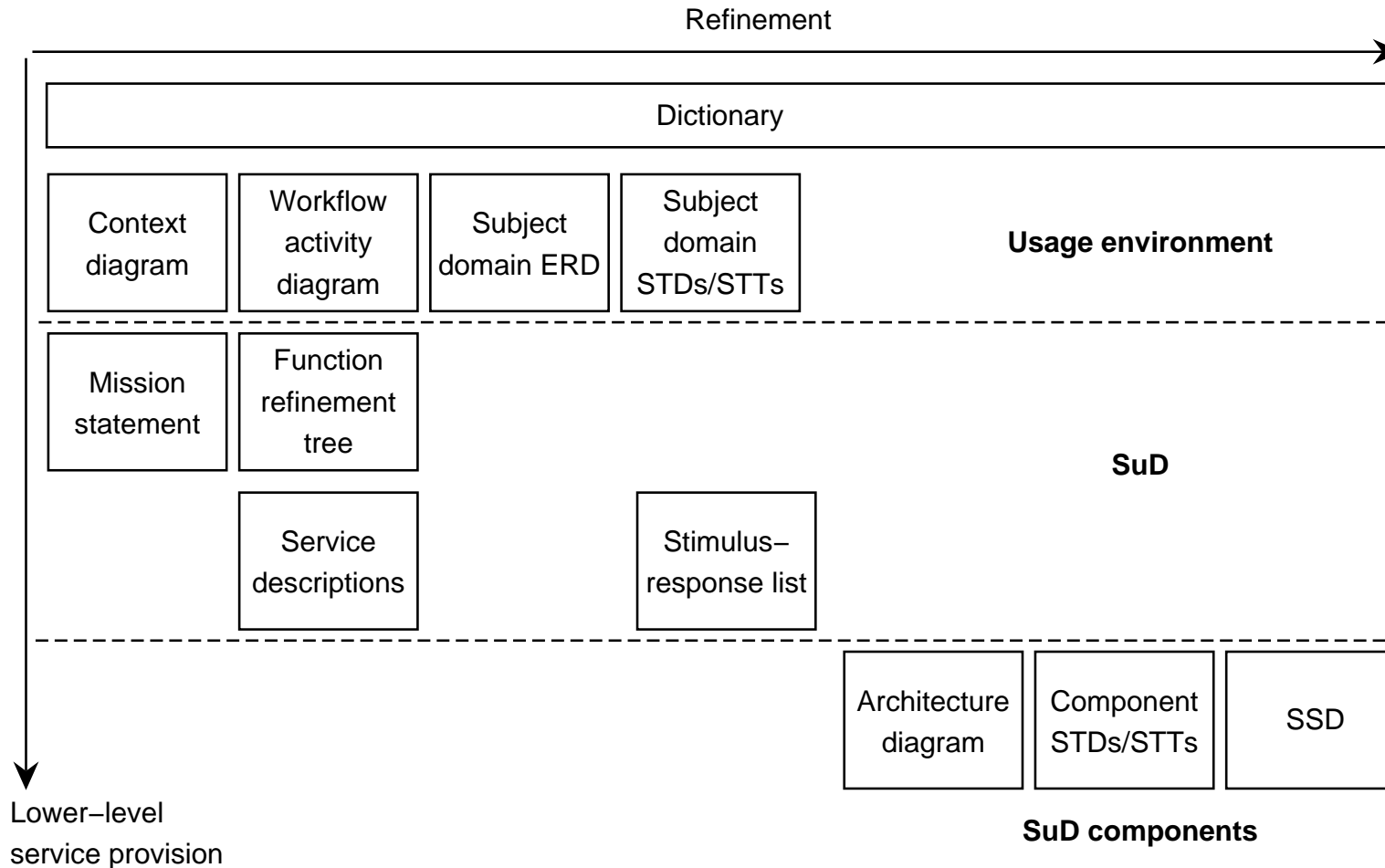
Notations treated in the book

	Func- tions	Beha- vior	Com- mu- nica- tion	De- com- posi- tion
Mission stmnt	X			
Funct. refinmnt tree	X			
Service descr.	X			
State trans. list		X		
State trans. table		X		
State trans. di- agr.		X		
Activity diagram		X		
Data flow diagr.			X	
Arch. diagram			X	
ERD				X
SSD				X

Our use of notations

Use environment of SuD	<ul style="list-style-type: none"> • Context diagram • Subject domain ERD • Activity diagram of user workflow • STT or STD of desired subj. dom. behavior • STT or STD of assumed subj. dom. behavior
Functional specification of SuD	<ul style="list-style-type: none"> • Mission statement • Function refinement tree • Service descriptions • Stimulus-response descriptions
Decomposition of SuD	<ul style="list-style-type: none"> • Architecture diagram • STT or STD of component behavior • Static structure diagram

Ordering of design decisions



Often, in order to make a lower-level design model, the designer has to first make a more refined higher-level description.

- *Problem bounding.* Environment modeling.
- *Service description.* System design.
- *Defining key terms.* Environment modeling.
- *Identifying desired and assumed behavior.* System design.
- *Decomposition.* System design.

Engineering arguments

- Engineers predict product properties from product specification.
- Tinkerers fiddle around with the product and discover how it behaves.

Feed forward versus feedback loop.

⇒ We use product descriptions, among others, to produce engineering arguments. Ranges from very informal to very formal.

Using engineering knowledge:

- Products with this kind of decomposition usually have properties P ;
- Since this product will have this kind of decomposition,
- It will probably have properties P .

Using throw-away prototyping:

- Since the prototype has properties P ,
- And the prototype is similar to the final product,
- The final product probably has properties P .

Using model execution:

- Since the model execution has properties P ,
- If the system implements this model exactly,
- Then the system will have properties P .

Using model checking:

- Since the state transition graph has properties P ,
- If the system implements this graph correctly,
- the system will have properties P .

Using theorem-proving:

- Since the decomposition has been proved to have properties P ,
- If the system correctly implements this decomposition,
- The system will have properties P .

Formality versus precision

- A description is **precise** if it expresses as briefly as possible what is intended.
- It is **formal** if it uses a language for which formal, meaning-preserving manipulation rules have been defined.

⇒

- Formal descriptions can be very imprecise: Statechart with superfluous transitions and states.
- Precise descriptions can be very informal: Function descriptions.

The attempt to be precise has priority over the attempt to be formal.

C.J. Smith, *Synonyms Discriminated*. G. Bell and Sons, Ltd., 1926.

- PRECISE denotes the quality of exact limitation, as distinguished from the vague, loose, doubtful, inaccurate; ... The idea of precision is that of casting aside the useless and superfluous.
- EXACTNESS is that kind of truth which consists in conformity to an external standard or measure, or has an internal correspondence with external requirement ... an exact amount is that which is required.”
- ACCURACY, by contrast, refers to the attention spent upon a thing, and the exactness which may be expected from it. Accuracy is designed whereas exactness may be coincidental.
- CORRECTNESS, finally, applies to what is conformable to a moral standard as well as to truth generally, as “correct behavior”.

C.J. Smith, *Synonyms Discriminated*. G. Bell and Sons, Ltd., 1926.

“It is most desirable that men should be exact in duties and obligations, accurate in statements and representations, correct in conduct, and precise in the use of words.”

Omit needless words

W. Strunk Jr. & E.B. White, *The Elements of Style*. Fourth edition. Allyn and Bacon 2000.

Rule 17

Vigorous writing is concise. A sentence should contain no unnecessary words, a paragraph no unnecessary sentences, for the same reason that a drawing should have no unnecessary lines and a machine no unnecessary parts.

This requires not that the writer make all sentences short, or avoid all detail and treat subjects only in outline, but that every word tell.