Formally Grounded Specifications Requirements Methodology

C. Choppy

LIPN, Université Paris XIII France

http://lipn.univ-paris13.fr/~choppy

Joint work with

G. Reggio

DISI, Università di Genova, Italy

Plan

- A formally grounded development method for formal requirements specifications
- Link with “less formal” use cases
- …

Outline and motivation

- Write relevant, legible, useful specifications of the systems to be developed
- Informal notations (graphics)/formal (semantics)
- Companion user method helping to understand the system to be developed (different from helping to use the proposed formalism)
- Accomodate different natures of systems
- The best of both worlds !?

	FORMAL	INFORMAL
notation	not very friendly (exotic)	very friendly, visual
notation	rigid, overhead	flexible, adaptable
learning	time, background	short(?)
case studies	simple (?)	real common app

Outline and motivation (2)

Methods taking into account:

- a software item:
  - *a simple dynamic system*
  - *a structured dynamic system*
  - *a data structure*
- two specification techniques: *property-oriented, model-oriented* (constructive)
- CASL and CASL-LTL specifications

Illustration on case studies

To be used

- for requirement specifications
- in combination with structuring concepts as (Jackson's) problem frames

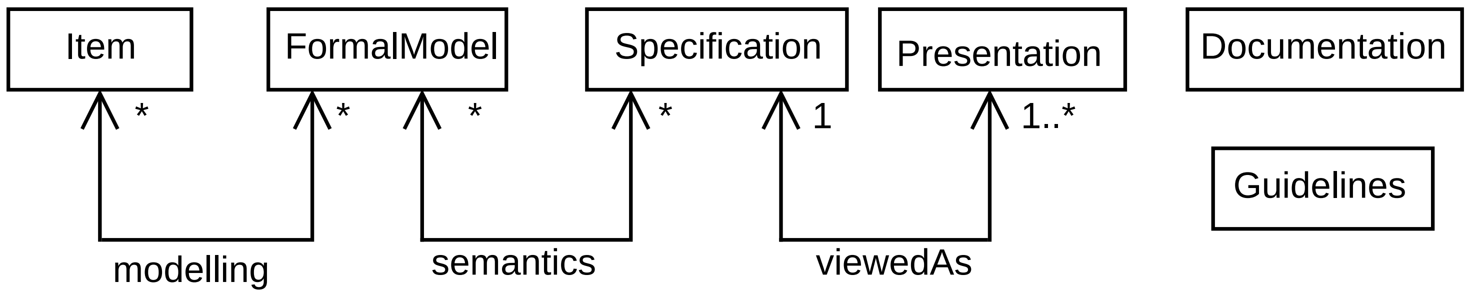
Case Study: a lift system

- a *lift plant* (the cabin, the motor moving it, the doors at the various floors)
- the *controller* (some software automatically controlling the lift functioning)
- the *users*

- *sensors* (e.g., cabin position, doors at floors, motor working status)
- *orders* (e.g., open/close the doors, move up/down/stop motor)
- users enter or leave the cabin . . .

Ingredients for a generic specification method

adapted from Astesiano, Reggio, TCS 2000.



1 - Items that will be specified

2 - Formal models of the items

3 - Modelling

4 - Specification

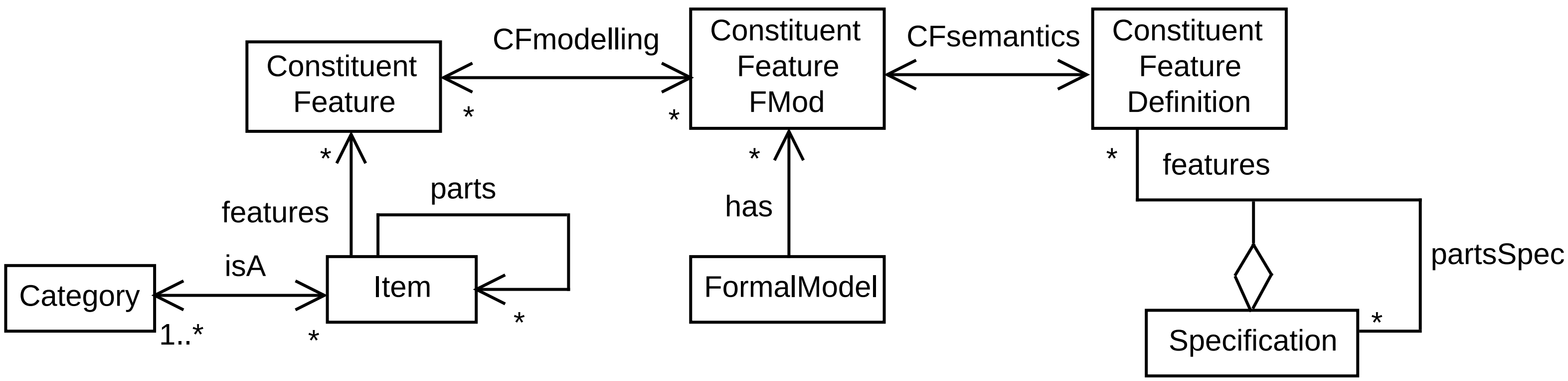
5 - Semantics

6 - Presentation

7 - Documentation

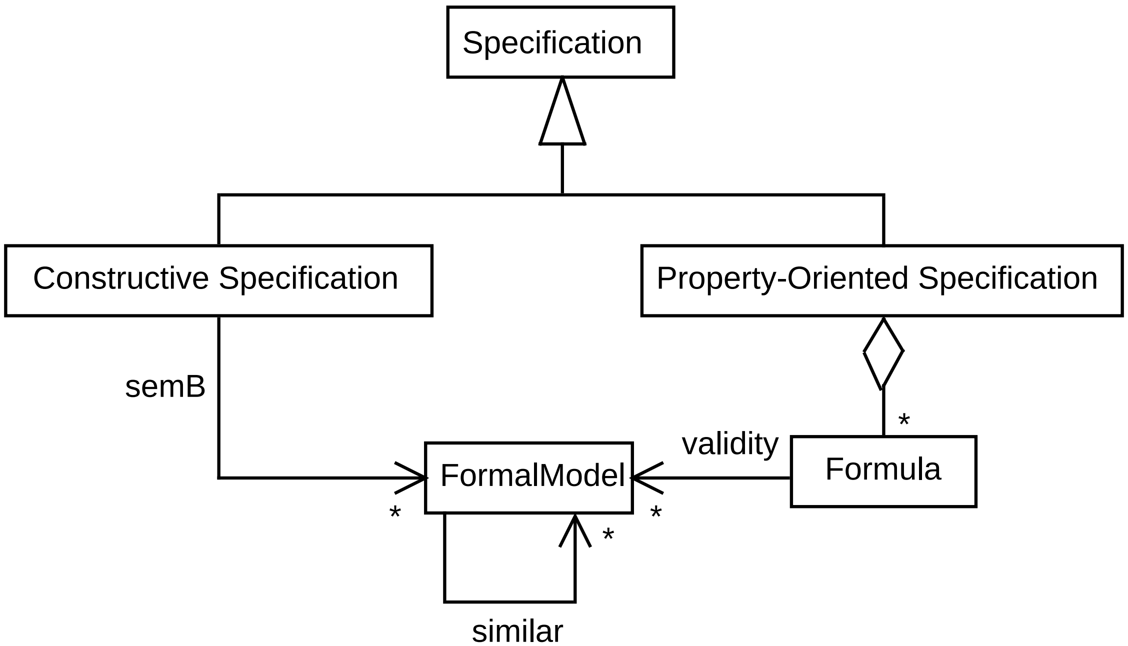
8 - Guidelines

Items



- *structured* (parts)
- characterized by constituent features of different *kinds*

Property vs Model oriented



- Property-oriented (axiomatic) : "relevant" properties expressed
- Model-oriented (constructive) : exhibit a prototype . . .

for: *simple dynamic systems, structured dynamic systems, data structures*

"6" specification methods with common parts.

CASL and CASL-LTL

- CASL (Common Algebraic Specification Language)

  partial ops, datatypes declarations, union, extension

  free construct, generic specifications

- CASL-LTL a simple system is considered as a labelled transition system (lts):

  labels, states and transition relation

  Labelled Transition Logic [Astesiano, Reggio, Costa, TCS97]

  dsort *st* **label** *lab* stands for **sorts** *st*, *lab*

  pred $\_\_ \xrightarrow{\_\_} \_\_ : st \times lab \times st$

  temporal logic (branching, CTL like) used to express properties of the

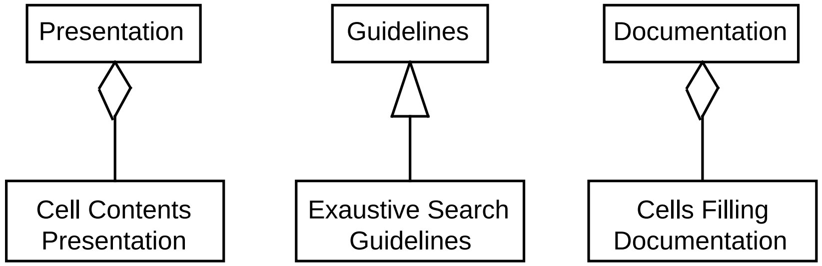
  dynamic systems in terms of their paths or sequences of transitions, e.g. :

  $in\_any\_case(S, \pi)$ or $in\_one\_case(S, \pi)$

  when a formula holds on the first state of a path,

  at the first label of a path, eventually, always . . . .

A General Property-oriented Specification Method (GPSm)



Find: parts, constituent features, express properties (cell filling, presentation).

		KIND$_1$			.....	KIND$_k$		
		CF^1_1	.....	CF^1_{n1}	.....	CF^k_1	.....	CF^k_{nk}
KIND$_1$	CF^1_1							
	.....							
	CF^1_{n1}							
	.....							
KIND$_k$	CF^k_1							
	.....							
	CF^k_{nk}							

Outline

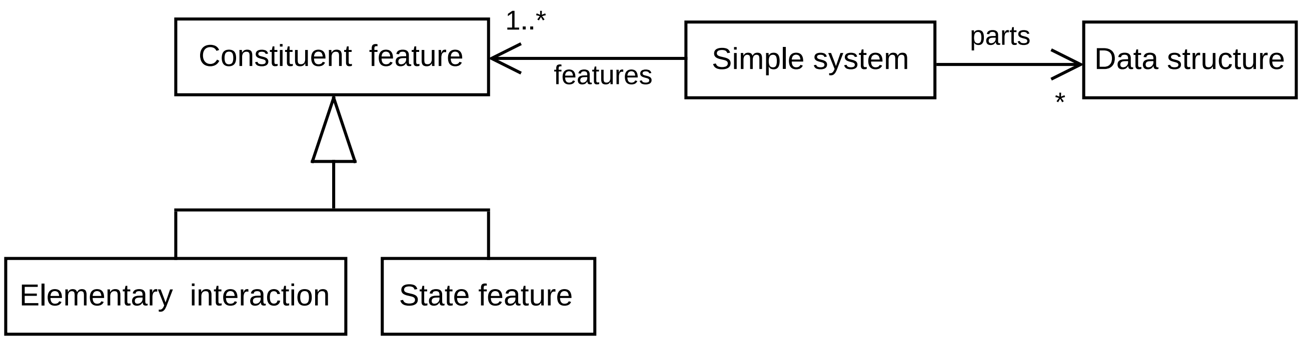
Methods taking into account:

- a software item:
  - a simple dynamic system
  - *a structured dynamic system*
  - *a data structure*
- two specification techniques: *property-oriented, model-oriented* (constructive)
- CASL and CASL-LTL specifications

Illustration on case studies

- in combination with structuring concepts as (Jackson's) problem frames

A Simple System



A dynamic system without any internal components cooperation.

A labelled transition system.

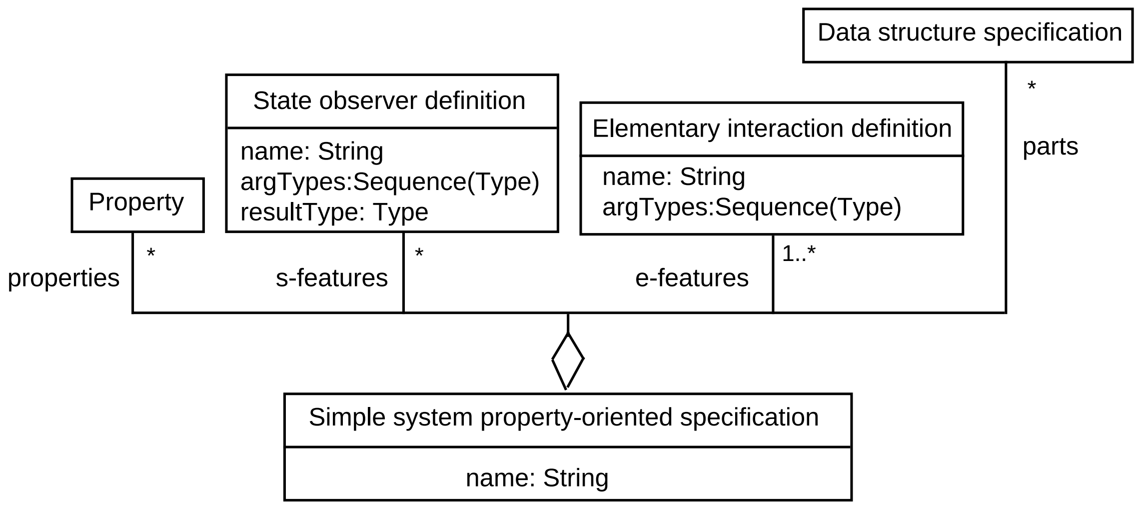
Constituent features:

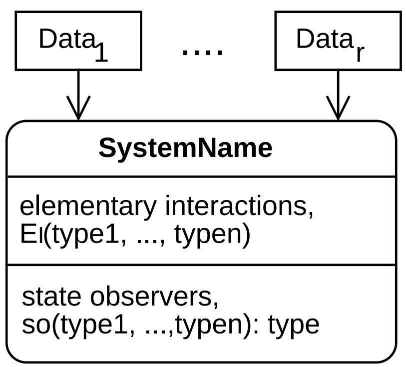
- state constituent features

- label: elementary interactions of different types

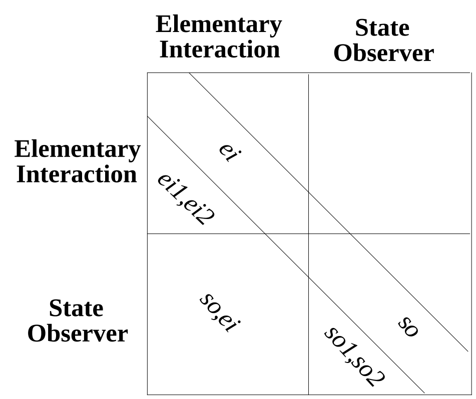
Parts: data structures

Property-oriented specifications (Simple systems)



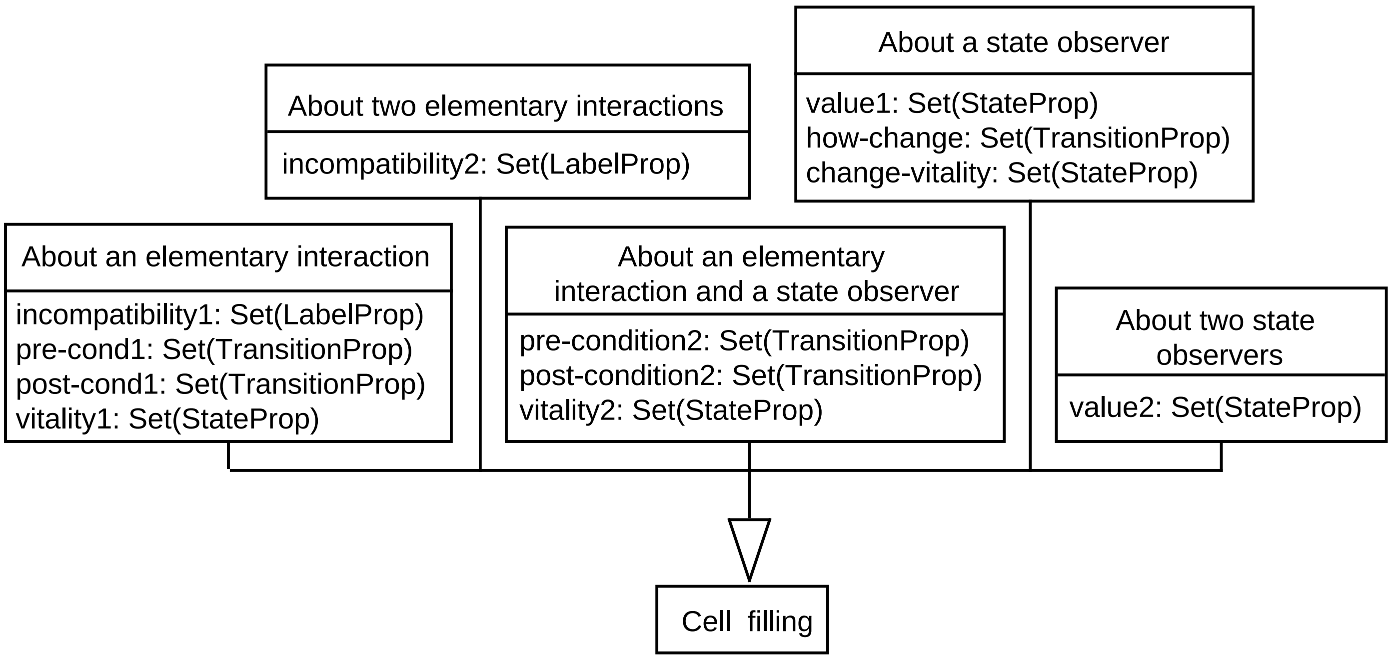


Visual presentation



Cell filling

Cell schemata (Simple system/Property-oriented)



Each cell may contain several properties of different nature. Properties on:

- labels (incompatibilities between elementary interactions under some condition)

- states (state observers properties where path properties may appear)

- transitions (conditions on source and target state observers).

Cell: About a state observer (*so*) -(Simple/Property)

value1 (state property) The results of the observation made by *so* on a state must satisfy some conditions.

cond, where *so* must appear in *cond*

how-change (transition property) If the observed value changes during a transition, then some condition on the source and target state (the old and the new value) holds, and some elementary interactions must belong to the transition label.

if *so*(*arg*) = v_1 **and** *so*'(*arg*) = v_2 **and** $v_1 \neq v_2$ **then** *cond*(v_1,v_2,*arg*) **and** $ei_1, \ldots, ei_n$ **happen**

change-vitality (state property) If a state satisfies some condition, then the observed value will change in the future.

if *cond*(v_1,v_2,*arg*) **and** *so*(*arg*) = v_1 **and** $v_1 \neq v_2$ **then in any case eventually** *so*(*arg*) = v_2

Note: “at least in a case” (instead of “in any case”) or “next” (instead of “eventually”) are possible.

Cell: About an elementary interaction (*ei*) -(Simple/Property)

incompatibility1 (label property) If their arguments satisfy some conditions, then two instantiations of *ei* are incompatible, i.e., no label may contain both.

$ei(arg_1)$ **incompatible with** $ei(arg_2)$ **if** $cond(arg_1, arg_2)$

pre-cond1 (transition property) If the label of a transition contains some instantiation of *ei*, then the source state of the transition must satisfy some condition.

if $ei(arg)$ **happen then** $cond(arg)$ where source state observers must appear in $cond(arg)$ and target state ones cannot appear

post-cond1 (transition property) If the label of a transition contains some instantiation of *ei*, then the target state of the transition must satisfy some condition). This may involve the source state.

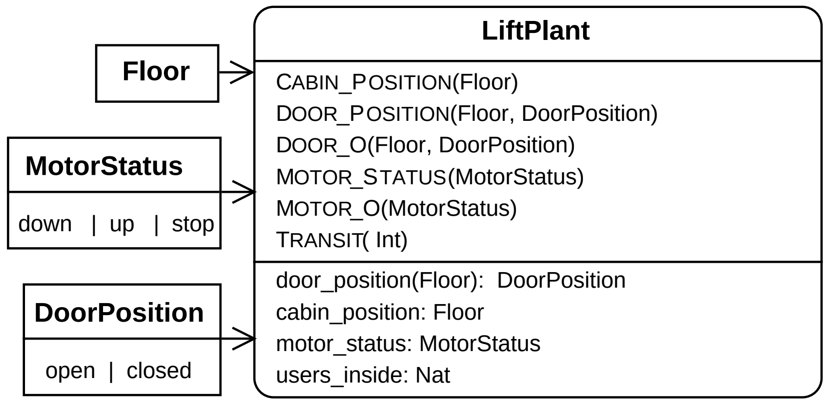
if $ei(arg)$ **happen then** $cond(arg)$ where target (primed) state observers must appear in $cond(arg)$ and source (non-primed) state ones may appear

Cell: About an elementary interaction (*ei*) - 2 (Simple/Property)

vitality1 (state property) If a state satisfies some condition, then any sequence of transitions starting from it will eventually contain a transition whose label contains *ei*. Note that vitality properties may have also the form “at least in a case” (instead of “in any case”) or “next” (instead of “eventually”).

if *cond*(*arg*) **then in any case eventually** *ei*(*arg*) **happen**

LiftPlant : Parts and Constituent Features (Simple/Property)



Parts: Floor, MotorStatus, DoorPosition

Constituent features

- *Elementary interactions*

CABIN_POSITION, DOOR_POSITION, DOOR_O, MOTOR_STATUS, MOTOR_O, TRANSIT

- *State observers*

door_position, *cabin_position*, *motor_status*, *users_inside*

Lift Plant properties - On MotorStatus (Simple/Property)

incompatibility1 (label property)

A sensor cannot signal two different values simultaneously.

MOTOR_STATUS(ms_1) **incompatible with** MOTOR_STATUS(ms_2) **if** $ms_1 \neq ms_2$

pre-cond1 (transition property)

A sensor always signals the correct data.

if MOTOR_STATUS(*ms*) **happen then** *motor_status* **=** *ms*

post-cond1 (transition property)

None

vitality1 (state property)

A sensor cannot break down, thus it may always be able to signal the correct value.

at least in one case next MOTOR_STATUS(*motor_status*) **happen**

Lift Plant properties - On the orders (Simple/Property)

Cell filling, drop repetition, rearrange, ...

- Only appropriate groups of orders may be received simultaneously by the lift plant; precisely at most one order for the motor and one for the doors.

MOTOR_O(*ms*$_1$) **incompatible with** MOTOR_O(*ms*$_2$) **if** *ms*$_1 \neq$ *ms*$_2$

DOOR_O(*f*$_1$,*dps*$_1$) **incompatible with** DOOR_O(*f*$_2$,*dps*$_2$) **if** ...

- An order cannot be received when its execution may be problematic; precisely move up (down) only when the motor is stopped and the cabin is not at the top (ground) floor, and open the door at f only when no door is open, the cabin is at floor f and the motor is stopped.

if MOTOR_O(*up*) **happen then** *motor_status* **=** *stop* **and** *cabin_position* $\neq$ *top*

if MOTOR_O(*down*) **happen then** *motor_status* **=** *stop* **and** *cabin_position* $\neq$ *ground*

if DOOR_O(*f*$_1$,*open*) **happen then**

(**for all** *f* • **if** *f* $\neq$ *f*$_1$ **then** *door_position*(*f*) $\neq$ *open*) **and** *cabin_position* **=** *f*$_1$ **and** *motor_status* **=** *stop*

- The orders are always correctly executed.

if MOTOR_O(*ms*) **happen then** *motor_status*$'$ **=** *ms*

if DOOR_O(*f*,*dps*) **happen then** *door_position*$'$(*f*) **=** *dps*

CASL, CASL-LTL view - (Simple/Property)

- *poSpec*.parts = $\{$*ds*$_1$, ..., *ds*$_j\}$ data structure specifications

  DS$_1$, ..., DS$_j$ are the CASL-LTL presentations of *ds*$_1$, ..., *ds*$_j$
- *poSpec*.e-features $= \{ei_1, \ldots, ei_n\}$ the elementary interactions
- *poSpec*.s-features $= \{so_1, \ldots, so_m\}$ the state observers

spec ELINTERACTION =

free type *elInteraction* $::=$

ei$_1$.name(*ei*$_1$.argTypes) $|$ ... $|$ *ei*$_n$.name(*ei*$_n$.argTypes)

spec *poSpec*.name =

FINITESET[ELINTERACTION] **and** DS$_1$ **and** ... **and** DS$_j$ **then**

dsort *st* **label** *FinSet*[*elInteraction*]

ops *so*$_1$.name : *st* $\times$ *so*$_1$.argTypes $\longrightarrow ?$ *so*$_1$.resType

...

so$_m$.name : *st* $\times$ *so*$_m$.argTypes $\longrightarrow$ *so*$_m$.resType

axioms

formulae corresponding to the cell fillings

CASL CASL-LTL view: properties (Simple/Property)

- transition properties

	expressed by
cond	$S \xrightarrow{l} S' \Rightarrow$ *cond*'

where *cond*' is obtained from *cond* by adding

- S as extra argument to each source (non-primed) state observer,

- S' as extra argument to each target (primed) state observer,

and by the following replacement

	replaced by
"*eInt* **happen**"	*eInt* $\in l$

CASL CASL-LTL view: properties (follwd) (Simple/Property)

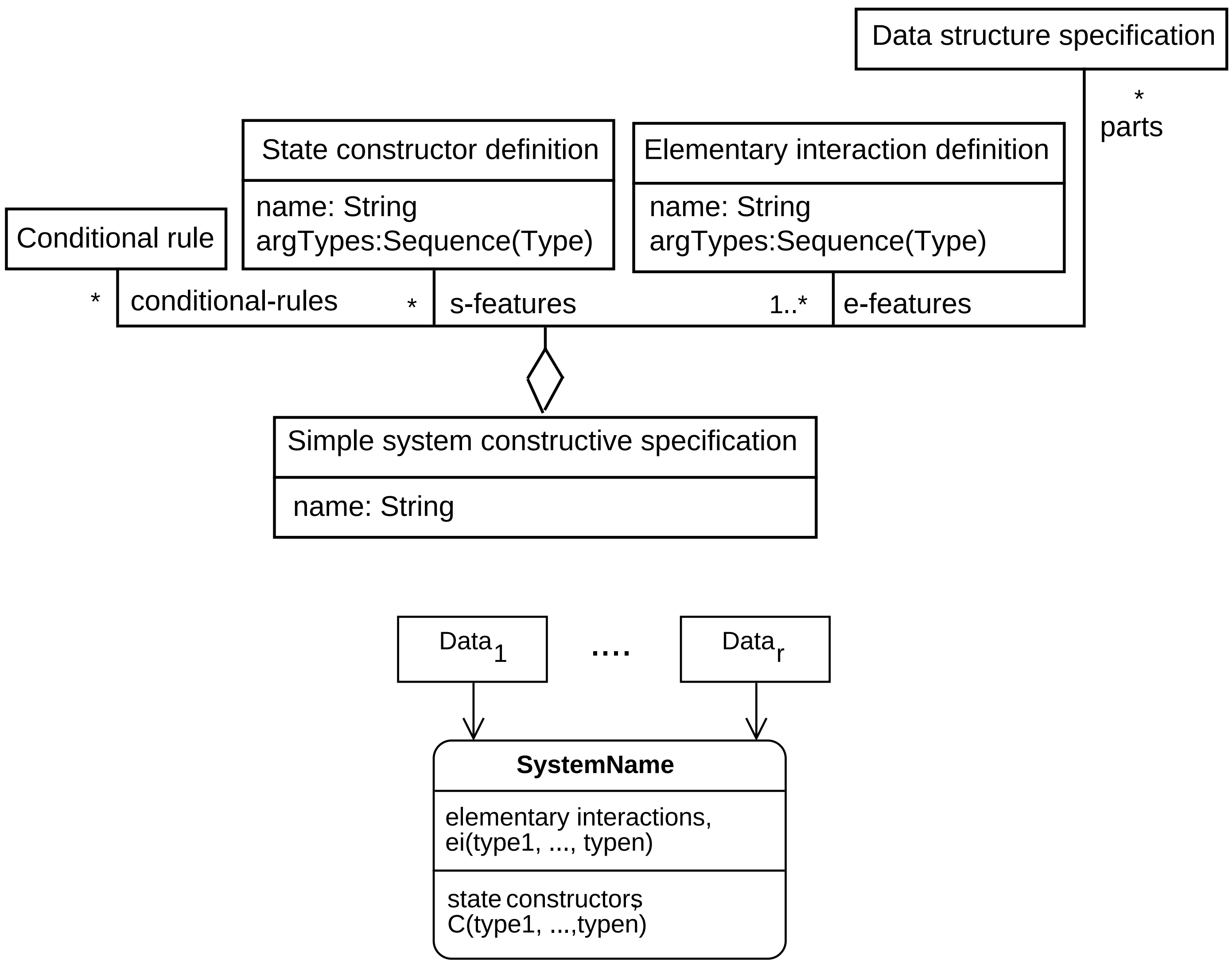
- label properties

eInt1 **incompatible with** *eInt2* **if** *cond*	*cond* $\Rightarrow \neg$ (*eInt1* $\in l \wedge$ *eInt2* $\in l$) var l: $FinSet$[*eInteraction*]

- state properties

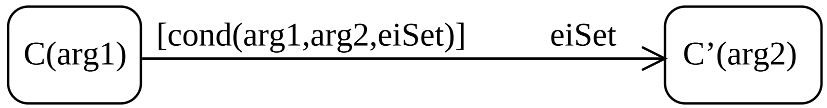
in any case ...	$in\_any\_case(S, \ldots)$
at least in one case ...	$in\_one\_case(S, \ldots)$
eventually *eInt*(arg) **happen**	$eventually < l \bullet$ *eInt*$(\mathrm{arg}) \in l >$
...	...

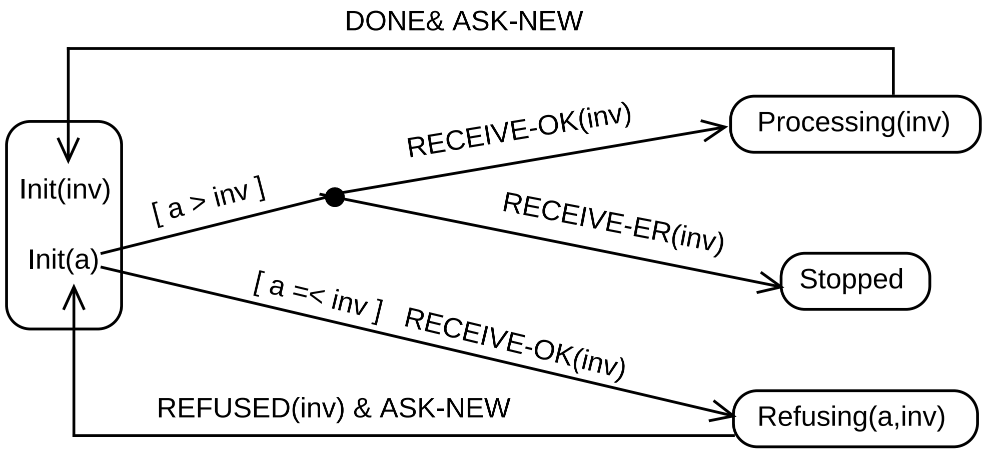
Constructive specifications (Simple systems)



Towards a Formally Grounded Development Method

Constructive specifications (Simple systems) - Properties





if $a > inv$ **then** $Init(a) \xrightarrow{RECEIVE-OK(inv)} Processing(inv)$

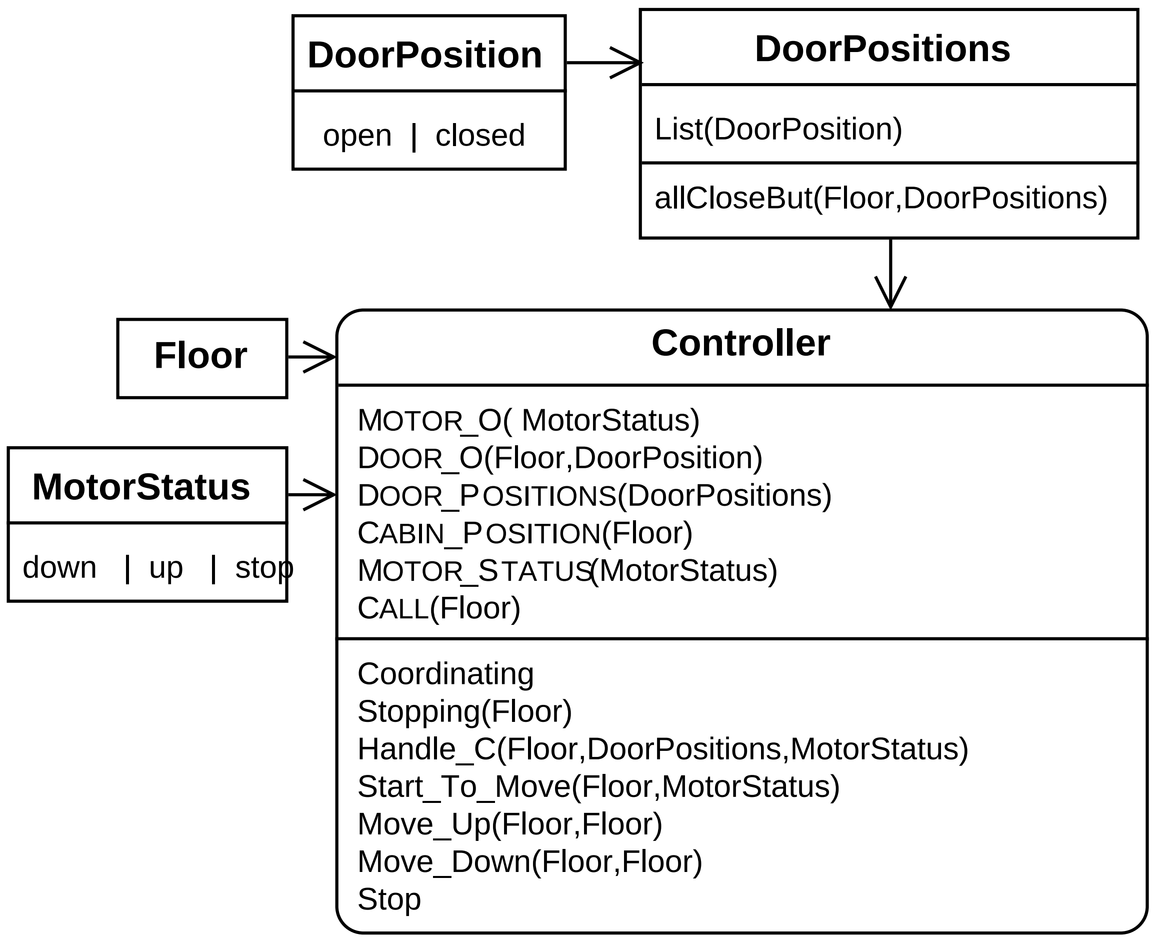
if $a > inv$ **then** $Init(a) \xrightarrow{RECEIVE-ER(inv)} Stopped$

if $a \leq inv$ **then** $Init(a) \xrightarrow{RECEIVE-OK(inv)} Refusing(a, inv)$

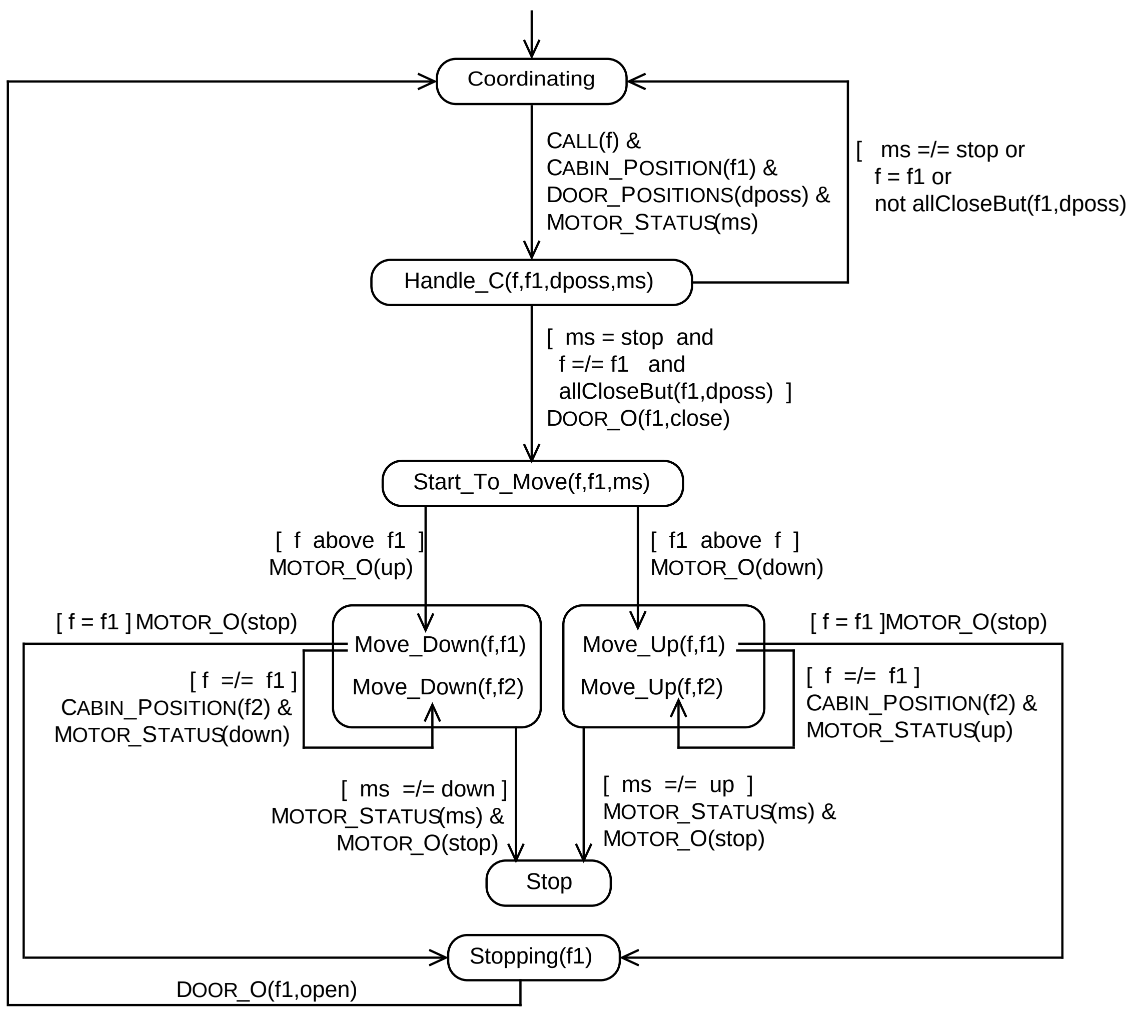
$$Refusing(a, inv) \xrightarrow{\{REFUSED(inv), ASK-NEW\}} Init(a)$$

$$Processing(inv) \xrightarrow{\{DONE, ASK-NEW\}} Init(inv)$$

Lift Controller (Simple/Constructive)



Lift Controller behaviour (Simple/Constructive)



Towards a Formally Grounded Development Method

CASL, CASL-LTL view: constructive spec of simple systems

- *conSpec*.parts = $\{$*ds* $_1$, . . . , *ds* $_j\}$

  DS $_1$, . . . , DS $_j$ are the CASL-LTL presentations of *ds* $_1$, . . . , *ds* $_j$
- *conSpec*.e-features $= \{ei_1, \ldots, ei_n\}$ the elementary interactions
- *conSpec*.s-features $= \{sCon_1, \ldots, sCon_m\}$ the state constructors

spec ELINTERACTION =

  free type *elInteraction* $::=$ *ei* $_1$.name(*ei* $_1$.argTypes) $|$ . . . $|$ *ei* $_n$.name(*ei* $_n$.argTypes)

spec *conSpec*.NAME =

  FINITESET[ELINTERACTION] **and** DS $_1$ **and** . . . **and** DS $_j$ **then**

free $\{$

  dsort *st* **label** *FinSet*[*elInteraction*]

  ops $sCon_1$.name : $sCon_1$.argTypes $\longrightarrow$ *st*

    . . .

    $sCon_m$.name : *st* $\times$ $sCon_m$.argTypes $\longrightarrow$ *st*

  axioms

    formulae corresponding to conditional rules

$\}$ **end**

Outline

Methods taking into account:

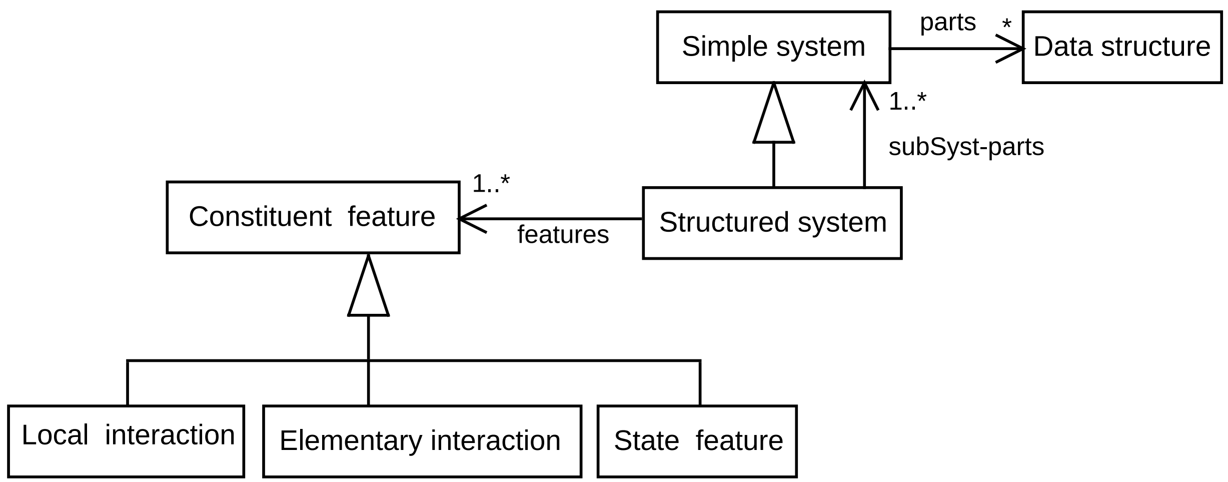
- a software item:
  - *a simple dynamic system*
  - a structured dynamic system
  - *a data structure*
- two specification techniques: *property-oriented, model-oriented* (constructive)
- CASL and CASL-LTL specifications

Illustration on case studies

- in combination with structuring concepts as (Jackson's) problem frames

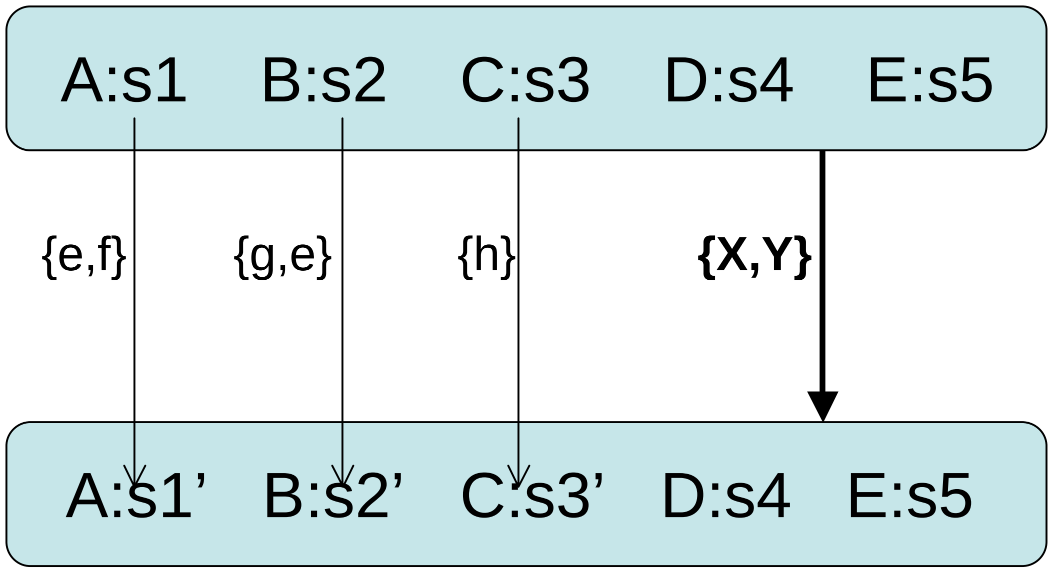
Structured Systems

- specialization of the simple dynamic systems
- simple or structured *subsystems* uniquely identified by some identity
- situation: subsystems situations
- global move: simultaneous/concurrent executions of subsystems (local) moves
- *generalized lts* - information: set of local moves

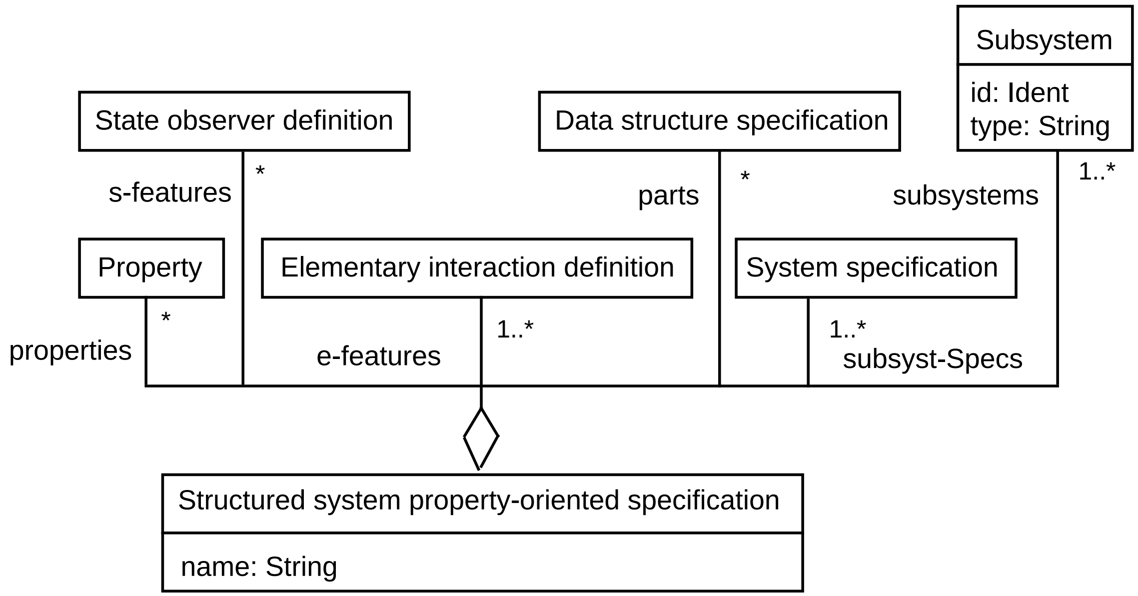


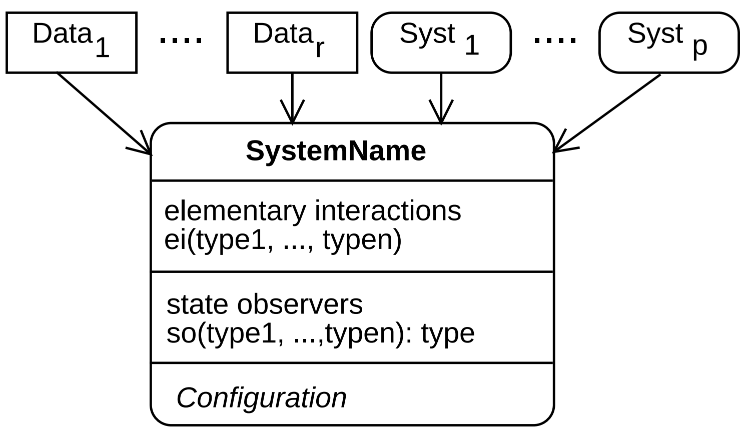
Transition of a structured system

- Local elementary interactions: A.e A.f …
- Global elementary interactions: X Y

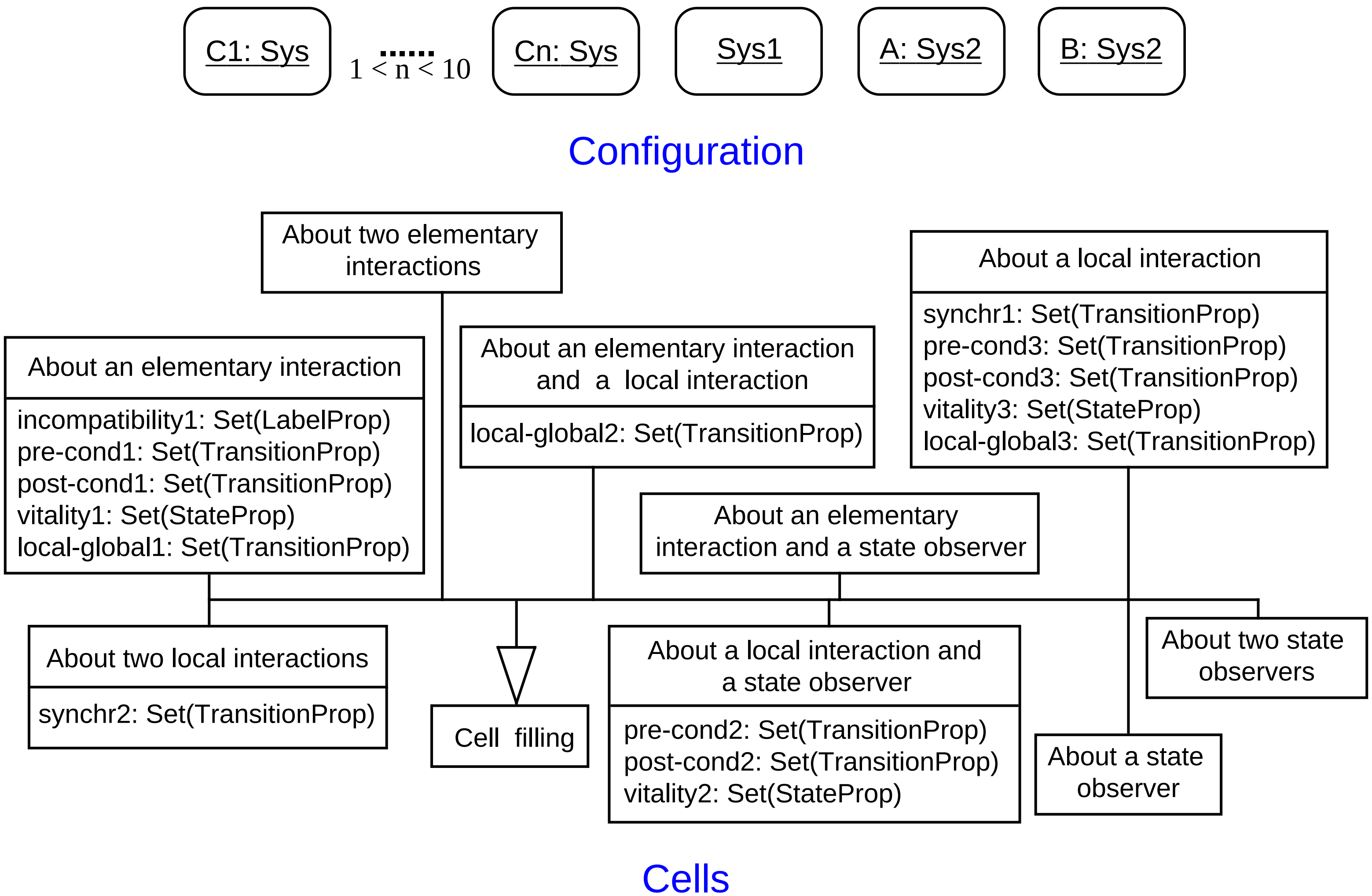


Property-oriented specifications of structured systems





Configuration and cells (Structured/Property)



Cell example About a local interaction : synchr1 and local-global3 (Structured/Property)

synchr1 (transition property)

An instantiation of the local interaction is synchronized (i.e., executed simultaneously)/not synchronized with another instantiation of the same; clearly the two instantiations are performed by different subsystems.

if *cond*(*arg*,arg_1) **and** *sid*.*ei*(*arg*) **happen then** sid_1.ei_1(arg_1) **happen**

or

if *cond*(*arg*,arg_1) **and** *sid*.*ei*(*arg*) **happen then not** sid_1.ei_1(arg_1) **happen**

local-global3 (transition property)

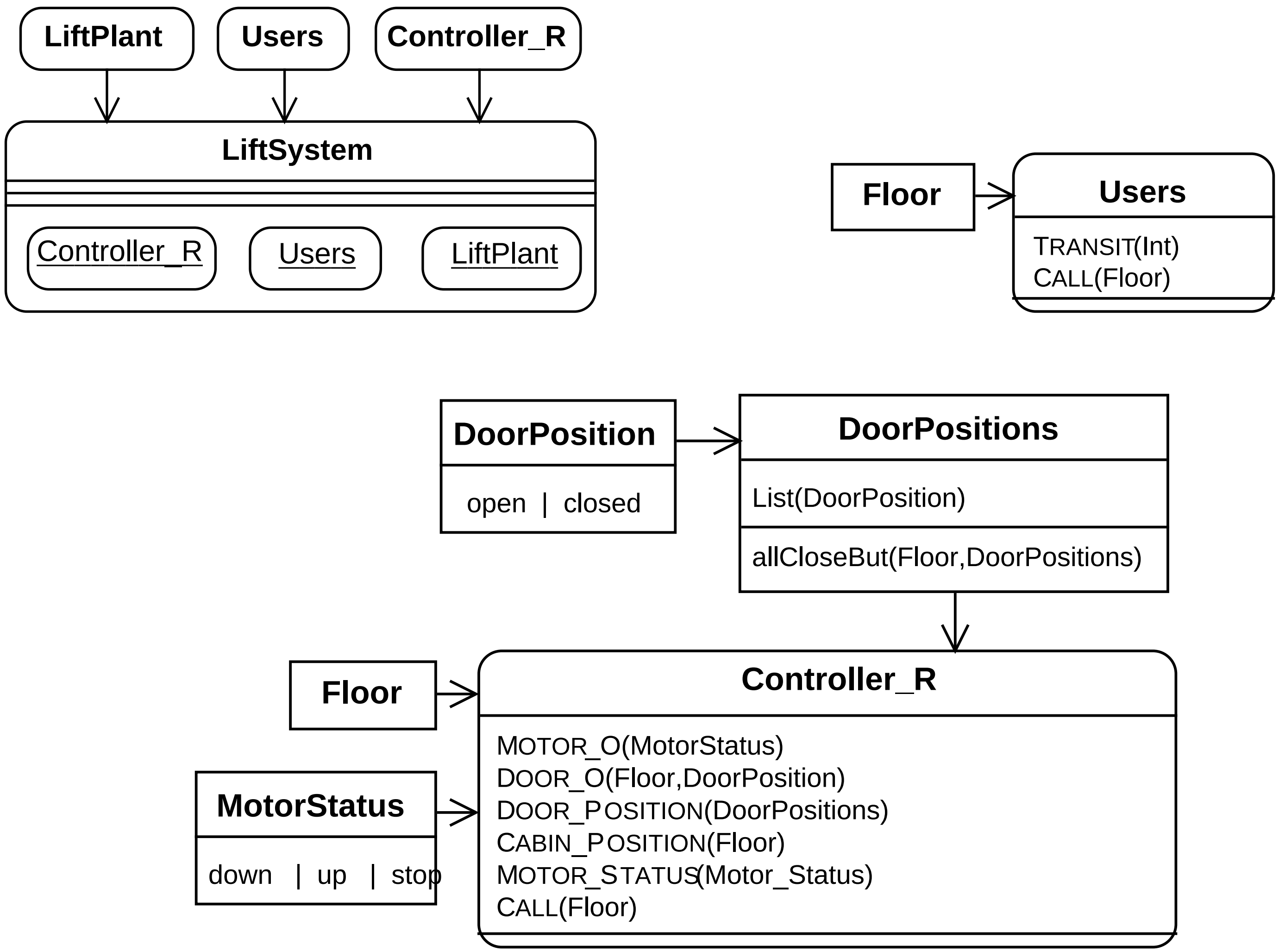
If an instantiation of *sid*.*ei* belongs to the label of some transition of some subsystem that is part of a global transition, then the label of such global transition must contain some elementary interaction, or vice versa.

if *sid*.*ei*(*arg*) **happen and** *cond*(*arg*,arg_1) **then** ei_1(arg_1) **happen**

or

if ei_1(arg_1) **happen and** *cond*(*arg*,arg_1) **then** *sid*.*ei*(*arg*) **happen**

Lift - Parts and Constituent Features (Structured/Property)



Lift Properties - (Structured/Property)

Local interactions with the same name and from different subsystems are synchronized

Users.CALL(f) **synchronized with** Controller_R.CALL(f)

LiftPlant.DOOR_POSITION(*ground*,dps_1), . . . LiftPlant.DOOR_POSITION(*top*,dps_{10})
synchronized with Controller_R.DOOR_POSITIONS(dps_1:: . . . :: dps_{10})

if Users.CALL(f) **happen then in any case eventually**

LiftPlant.*cabin_position*(f) **and**

LiftPlant.*motor_status*(*stop*) **and**

LiftPlant.*door_position*(f) **=** *open*

CASL-LTL View (Structured/Property)

dsort *st* **label** *lab* **info** *inf* stands for **sorts** *st*, *lab*, *inf*
pred $\_\_ : \_\_ \xrightarrow{\_\_} \_\_ : inf \times st \times lab \times st$

- *poSpec*.parts = $\{ds_1, \ldots, ds_j\}$, and that $\mathrm{DS}_1, \ldots, \mathrm{DS}_j$ are the CASL-LTL presentations of the data structure specifications $ds_1, \ldots, ds_j$ respectively
- *poSpec*.subsyst-Specs $= \{ssp_1, \ldots, ssp_k\}$, that $\mathrm{SSP}_1, \ldots, \mathrm{SSP}_k$ are the CASL-LTL presentations of the system specifications $ssp_1, \ldots, ssp_k$ respectively, and that $\mathrm{ElInteraction}_1, \ldots, \mathrm{ElInteraction}_k$ be the specifications of their elementary interactions.
- *poSpec*.e-features $= \{ei_1, \ldots, ei_n\}$ the elementary interactions
- *poSpec*.s-features $= \{so_1, \ldots, so_m\}$ the state observers
- *poSpec*.subsystems $= \{ss_1, \ldots, ss_r\}$ the subsystems

CASL-LTL View foll'd (Structured/Property)

spec LOCALINTERACTION =

ELINTERACTION$_1$ **and** ... **and** ELINTERACTION$_k$ **and** Ident **then**

free type *subElInteraction* $::=$ $\_(\textit{elInteraction}_1) \mid \ldots \mid \_(\textit{elInteraction}_k)$

%% disjoint union of the elementary interaction types of the subsystems

free type *localInteraction* $::= \; < \_, \_ > (\textit{ident}, \textit{subElInteraction})$

spec *poSpec*.name =

FINITESET[ELINTERACTION] **and** FINITESET[LOCALINTERACTION] **and**

DS$_1$ **and** ... **and** DS$_j$ **and** SSP$_1$ **and** ... **and** SSP$_k$ **then**

dsort *st* **label** *FinSet*[*elInteraction*] **info** *FinSet*[*localInteraction*]

ops so_1.name : $st \times so_1$.argTypes $\longrightarrow so_1$.resType **%%** state observers

...

so_m.name : $st \times so_m$.argTypes $\longrightarrow so_m$.resType

ss_1.id : $st \longrightarrow ss_1$.type **%%** observers of the subsystem states

...

ss_r.id : $st \longrightarrow ss_r$.type

axioms *those formulae corresponding to the cell fillings*

Outline

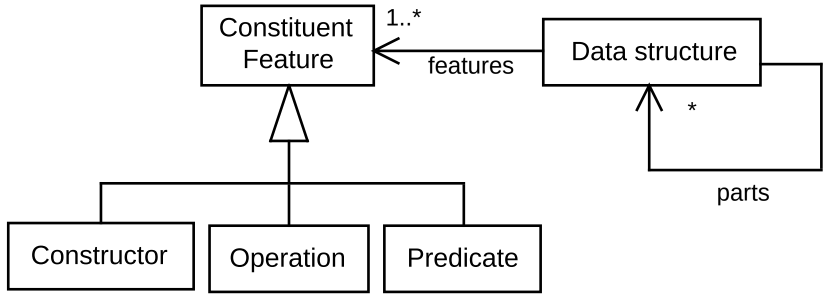
Methods taking into account:

- a software item:
  - *a simple dynamic system*
  - *a structured dynamic system*
  - a data structure
- two specification techniques: *property-oriented, model-oriented* (constructive)
- CASL and CASL-LTL specifications

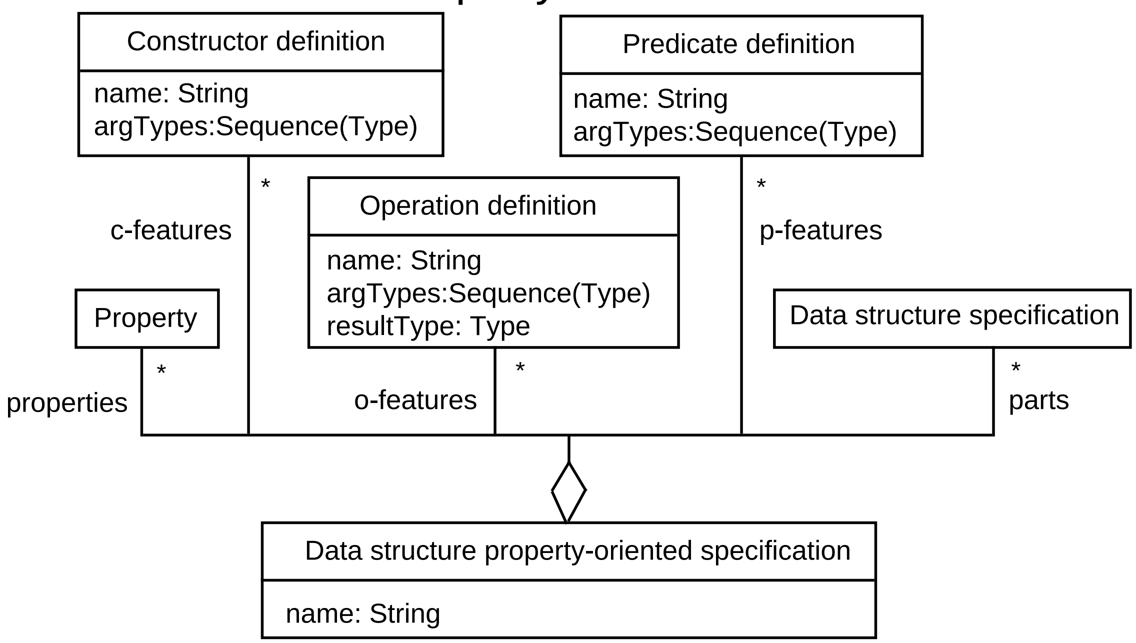
Illustration on case studies

- in combination with structuring concepts as (Jackson's) problem frames

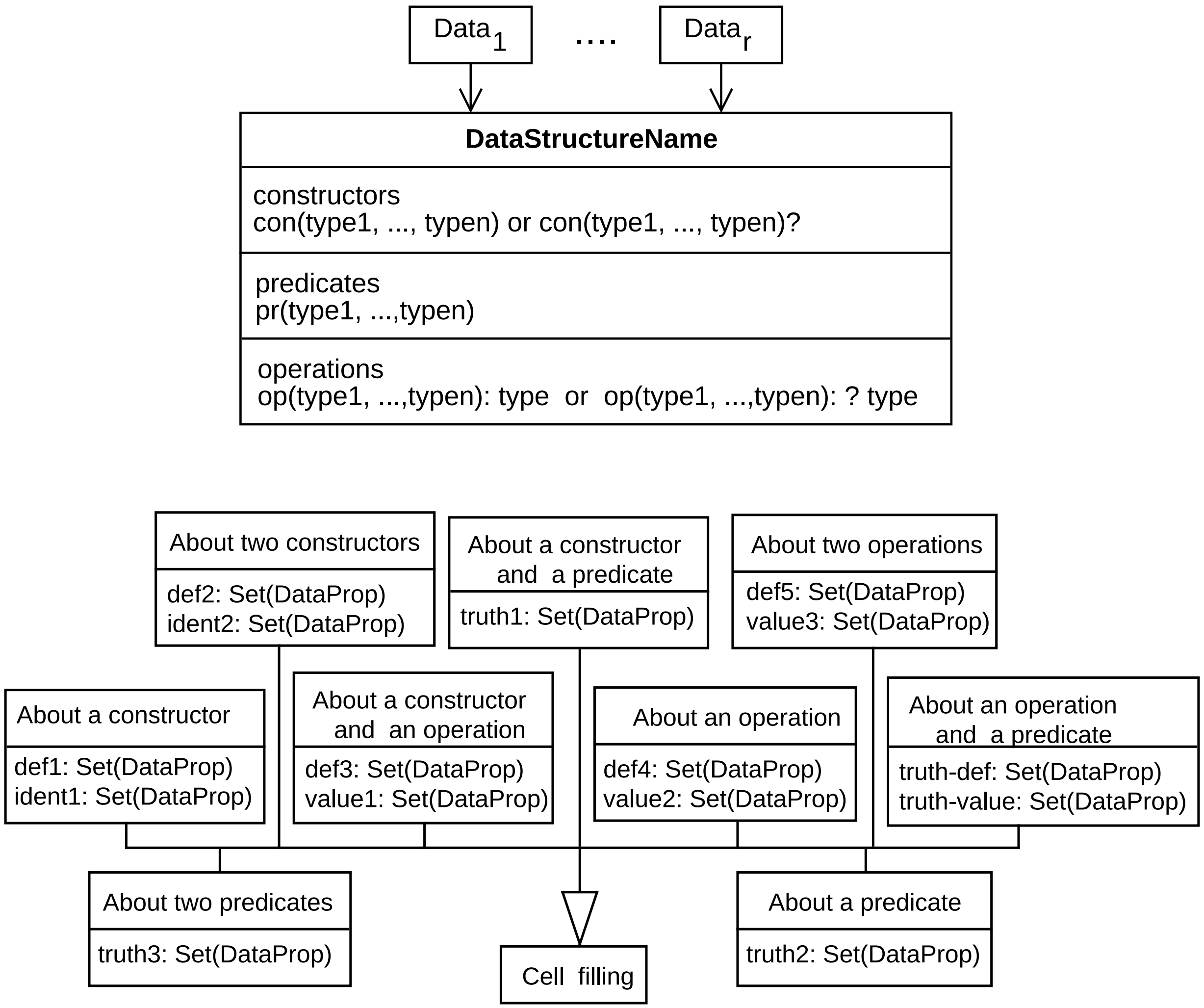
Data Structure Items / Property



Property-oriented



Data Structure - Property-oriented



Towards a Formally Grounded Development Method

Floor (Data Structure/Property-oriented)

Floor
ground top
_ above _(Floor,Floor)
next(Floor): ? Floor previous(Floor): ? Floor

– There exists a ground and a top floor, and they are different.

$ground \neq top$

– *next* returns the floor immediately above a given one, if it exists. There is no floor between f and $next(f)$.

def(*next*(*ground*))

not def(*next*(*top*))

def(*next*(*f*)) **iff** *top above f*

whenever everything is defined

next(*f*) *above f* **and not exists** $f_1 \bullet$ (*next*(*f*) *above* f_1 **and** f_1 *above f*)

whenever everything is defined *next*(*previous*(*f*)) **=** *previous*(*next*(*f*)) **=** *f*

– *above* is total order over the floors with *top* as maximum and *ground* as minimum . . . .

CASL View (Data/Property)

- *poSpec*.parts = $\{ds_1, \ldots, ds_j\}$ w/ $\mathrm{DS}_1, \ldots, \mathrm{DS}_j$ CASL-LTL presentations
- *poSpec*.c-features = $\{con_1, \ldots, con_n\}$ the constructors
- *poSpec*.o-features = $\{op_1, \ldots, op_m\}$ the operations
- *poSpec*.p-features = $\{pr_1, \ldots, pr_p\}$ the predicates.

spec *poSpec*.name =

DS_1 **and** ... **and** DS_j **then**

type *poSpec*.name $::=$ con_1.name(con_1.argTypes)? $|$ ... $|$ con_n.name(con_n.argTypes)

ops op_1.name : op_1.argTypes $\longrightarrow ?$ op_1.resType

...

op_m.name : op_m.argTypes $\longrightarrow$ op_m.resType

preds pr_1.name : pr_1.argTypes

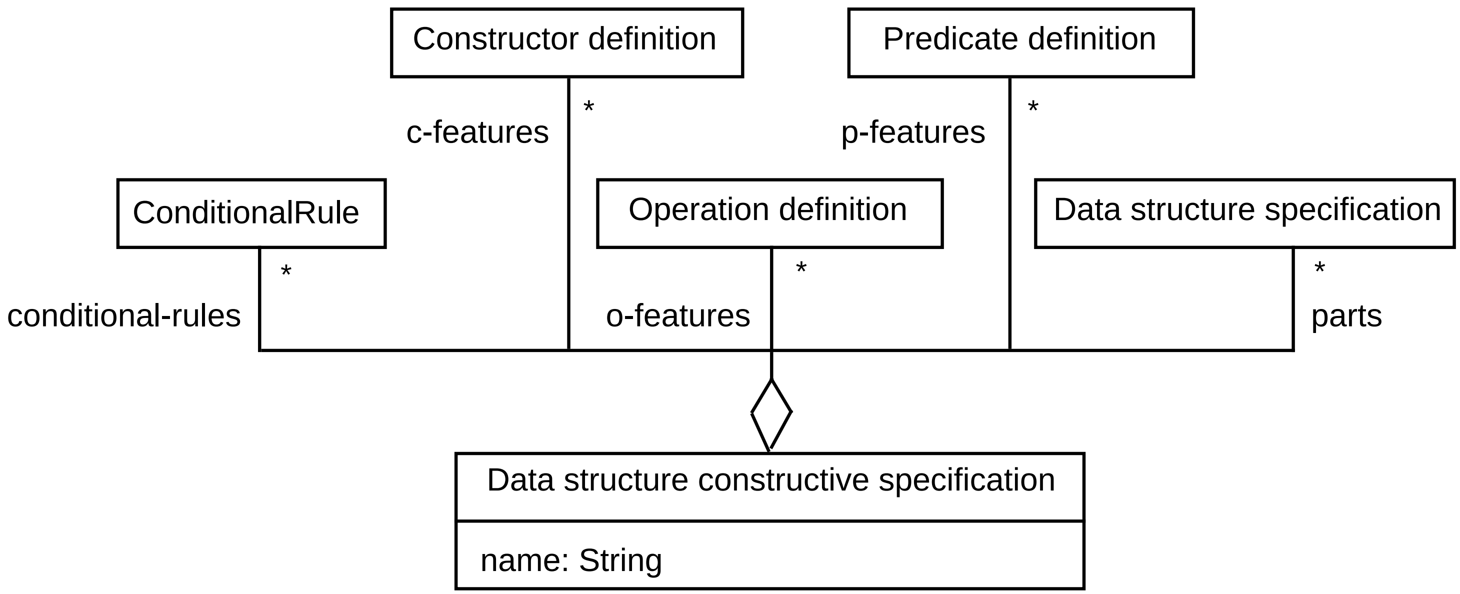
...

pr_p.name : pr_p.argTypes

axioms

formulae corresponding to the cell fillings

Data Structure - Constructive



Outline

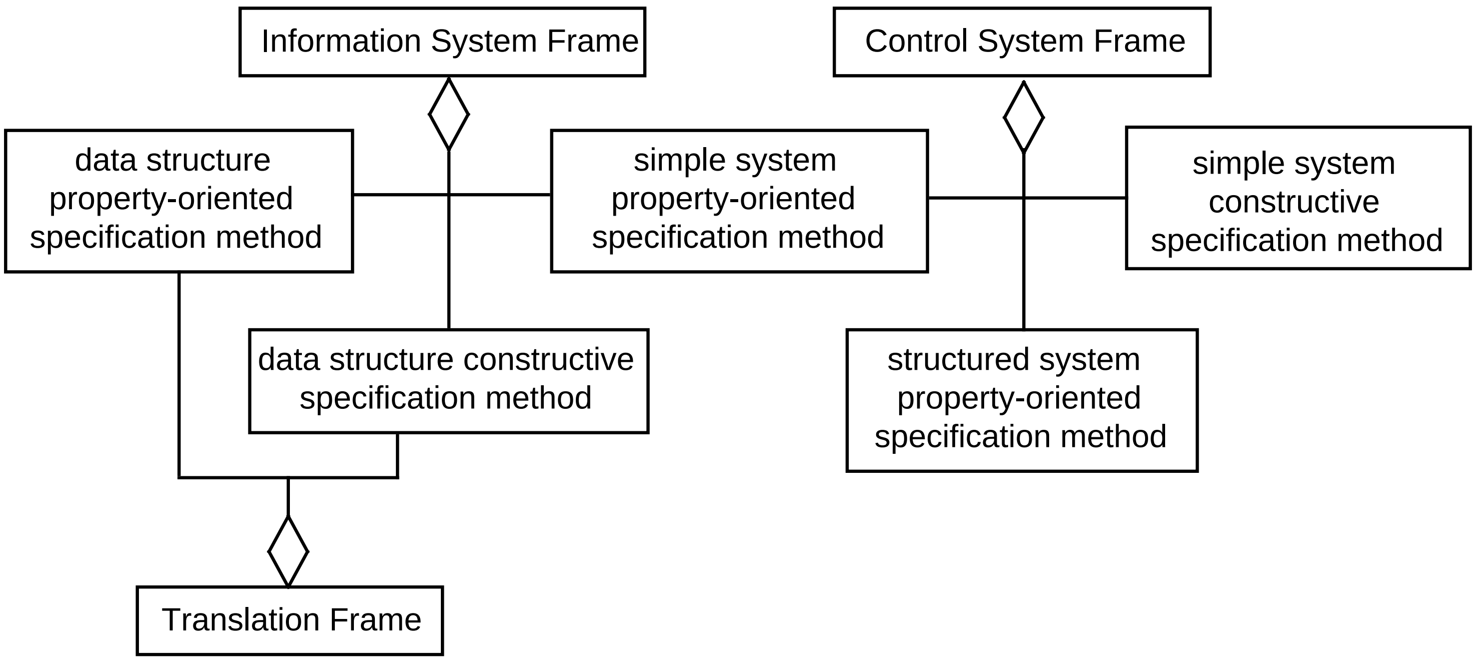
Methods taking into account:

- a software item:
  - *a simple dynamic system*
  - *a structured dynamic system*
  - *a data structure*
- two specification techniques: *property-oriented, model-oriented* (constructive)
- CASL and CASL-LTL specifications

Illustration on case studies

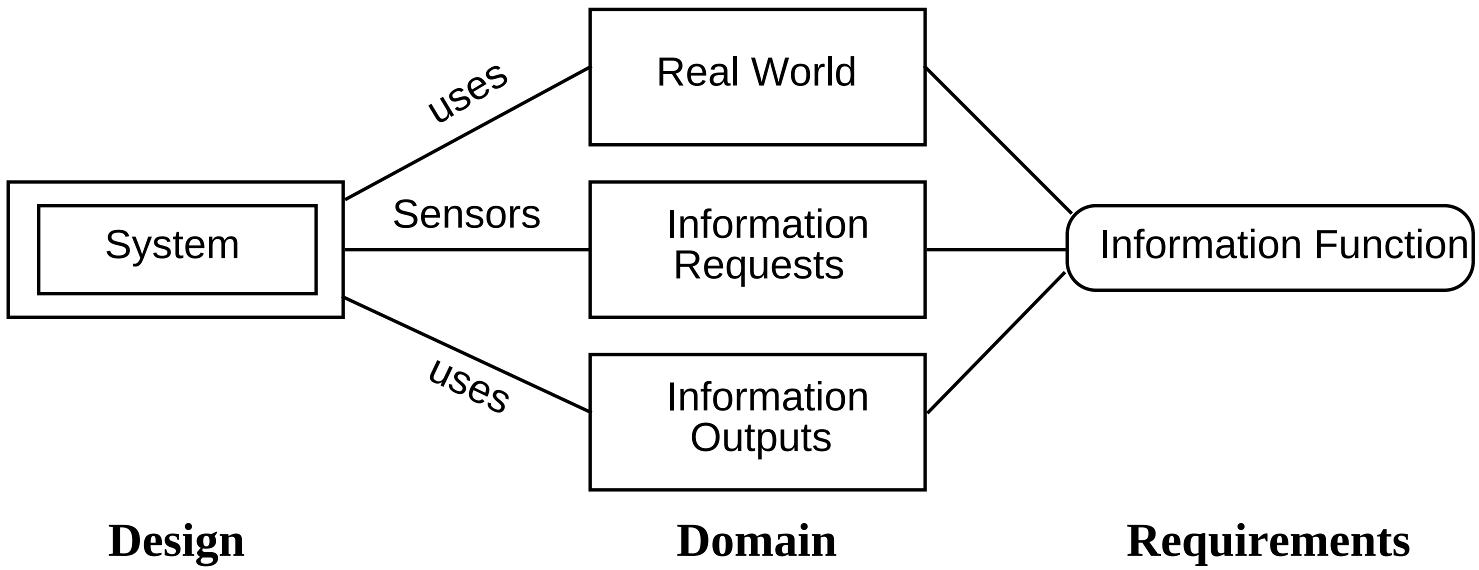
- in combination with structuring concepts as (Jackson's) problem frames

Applying our Specification Methods to Classes of Systems
(“Problem Frames”)



Towards a Formally Grounded Development Method

Information System Frame



Real World : simple dynamic system (property), signals relevant information

Information Requests/Outputs : data structure (model/constructive)

Information function : with a (model/constructive) data structure (History, ...)

System : simple system (model/constructive)

Conclusion and . . .

Companion method for (algebraic) formal specifications

- Paradigms, techniques, pragmatic characteristics originated from the underlying theory (e.g. no “use cases” . . . , no OO)
- both visual and explicit presentations
- systematic and inherently rigorous, cell-filling
- well defined underlying formal models
- experimented on sizeable case studies, on students
- “building-bricks” specification tasks for different kinds of software (simple systems, structured, data structures), at different abstraction level (property/more abstract, model or constructive/more concrete)
- relevant for real applications, used for requirement specifications, or in connection with structuring concepts (problem frames)

. . . Perspectives

- Our cell-filling technique can be a basis for generating precise UML models, or for their inspection (checking all aspects considered)
- Further experiments, new problem frames (business automation, web applications, distributed mobile systems, . . .)
- Oriented towards CASL and CASL-LTL (algebraic specifications) but adaptable to other specification/description paradigms
- Supporting tools (graphical editor, type checker, guidelines support, . . .)

Part II

Integrate the specification development method together with use cases requirement description

- Some description is required before a specification may be written

- Is it possible to establish a connection between both ?

- Guidelines for these tasks

HOW ? (1)

- Use USE CASES
  - use case =
    - description of interactions between the system under discussion and external actors, related to the goal of one particular actor
    - description is textual (“familiar”, easy to read) and sums up a set of scenarios (sequences of interactions between system and actors)
  - quite successful
    - easy to use and informal
    - easily give an idea of the system that can be discussed with the client
    - a lot of freedom in what should include a use case description, and how it should be written
  - however
    - “use cases are wonderful but confusing'' (Cockburn 2000)
    - use cases are often imprecise, and used terms are vague or ambiguous

HOW ? (2)

- Use formal specifications
  - lead to precise, unambiguous descriptions
  - but difficult to use and impractical in quite a number of cases
  - hard to write/read these specifications
  - hard to start with formal specifications while still working on the requirements (thus, trying to understand what is the problem about)

HOW ? (3)

- combine advantages of use cases and of formal specifications
  - improving use case based requirements by developing a companion Formally Grounded specification [ChoppyReggio2003]
    - written in a “visual” notation (diagrams and text)
    - with a formal counterpart written in the logical-algebraic CASL-LTL specification language
    - produced following a systematic method, arising questions on all the aspects of the specified system
  - resulting in
    - *requirement validation*, writing the Formally Grounded specification leads to thoroughly check that requirements
    - improved *requirements* (requirements may be updated)
    - improved use case based requirement specification
    - a formal specification available for *formal analysis*

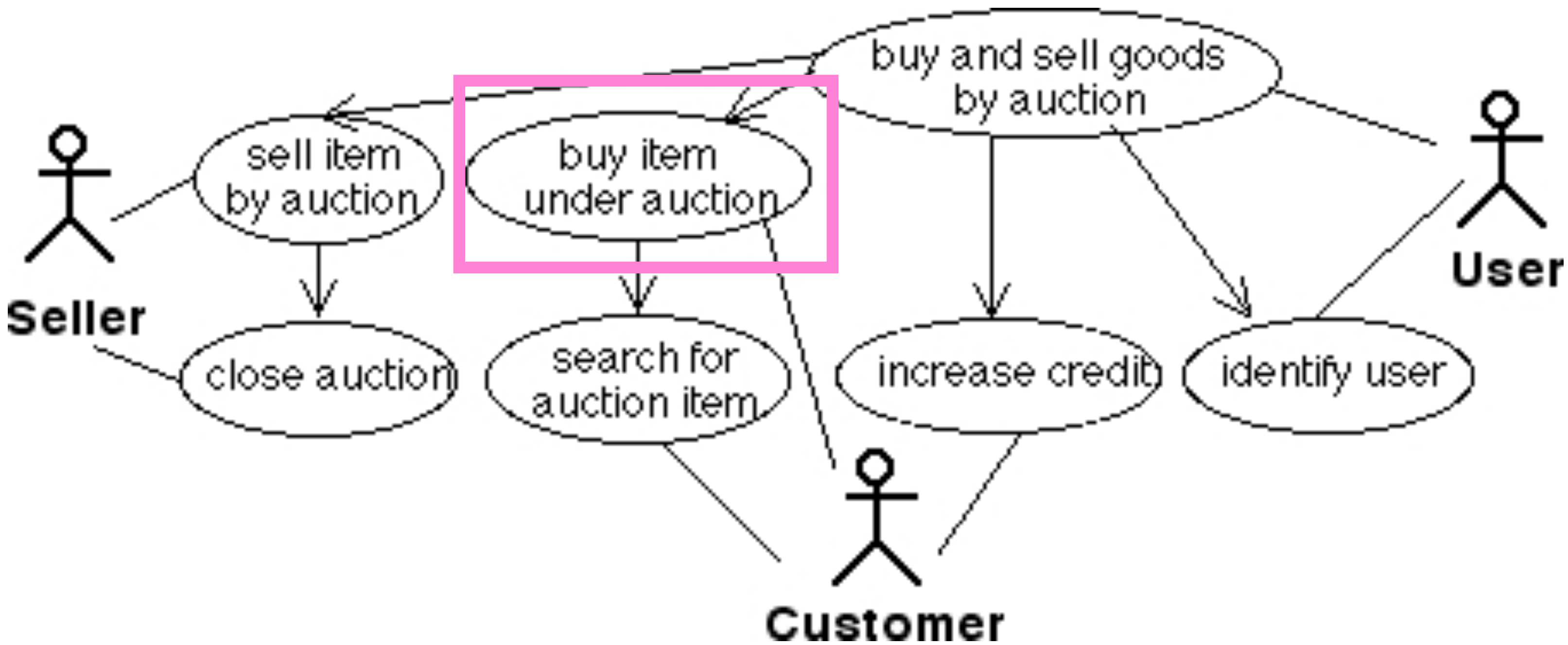
Case study: Auction System

- Online auction system to allow to buy/sell goods
- Innovative because it guarantees that bid placed are solvent
- Users must first enroll and log on for each session, then they are able to sell, buy, or browse the available auctions
- Customers have credit with the system used as security on each bid; and can increase it by asking the system to debit a certain amount from their credit card, and when sell
- A customer that wishes to sell initiates an auction by informing the system of the goods to auction with ….
- Customers that wish to follow an auction must first join the auction, then they may make a bid, or post a message
- Bidders are allowed to place their bids until the auction closes, and place bids across as many auctions as they please

Auction System: task 1

Give a Use case based requirement specification

- (UML) Use case diagram



- Use case descriptions
  (S. Sendall and A. Strohmeier template)

Use Case buy item under auction

Intention in Context: The intention of the Customer is to follow the auction, ...

Primary Actor: Customer

Precondition: Customer already identified

Main Success Scenario:

1. Customer searches for an item under auction (search item).
2. Customer requests to join the item auction.
3. System presents a view of the auction

...

buy item under auction (contd)

Steps 4-5 can be repeated according to the intentions and bidding policy of the Customer

4. Customer makes a bid on the item to System.
5. System validates the bid, records it, secures the bid amount from Customer's credit, … informs Participants of new high bid, and updates the view of the auction
6. System closes the auction with a winning bid by Customer.

Extensions: ...

buy item under auction (extens.)

Extensions:

2a. C requests System not to pursue item further:

2a.1. System permits Customer to choose another auction, or go back to an earlier point in the selection process; uc continues at step 2.

3a. System informs Customer that auction has not started: use case ends in failure.

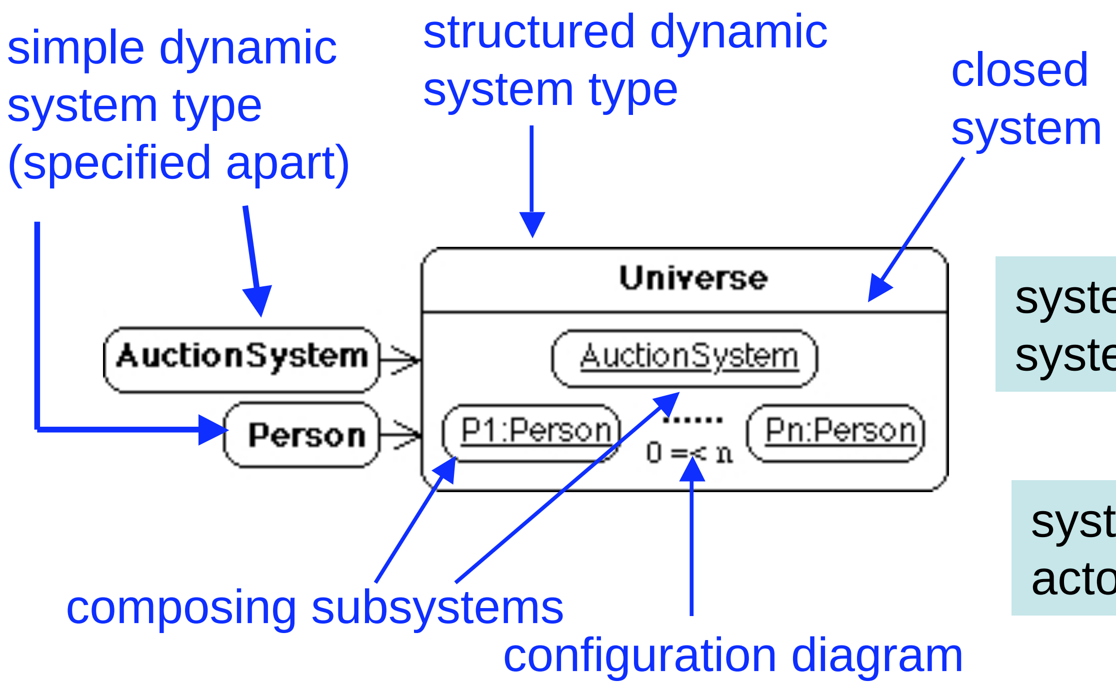
3b. System informs Customer that auction is closed: use case ends in failure.

...

Auction System: task 2

By looking at the Use case diagram give

- Context View (initial version)



system of type AuctionSystem = system to develop

systems of type Person = actors

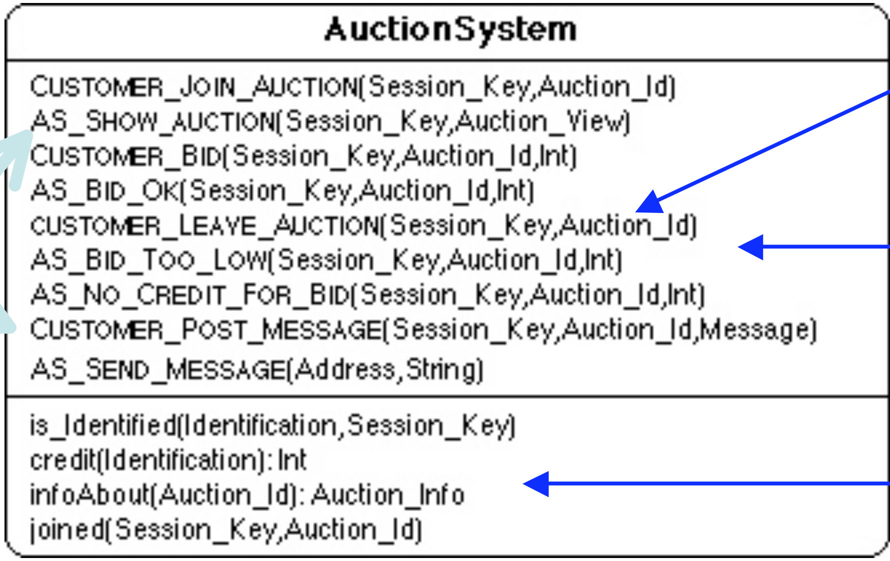
Auction System: task 3

By looking at use case descriptions one after the other (here Buy Item under Auction) give

- AuctionSystem specification interface

Data type

interactions towards actors or received by actors



Elementary interactions

Tested or updated information

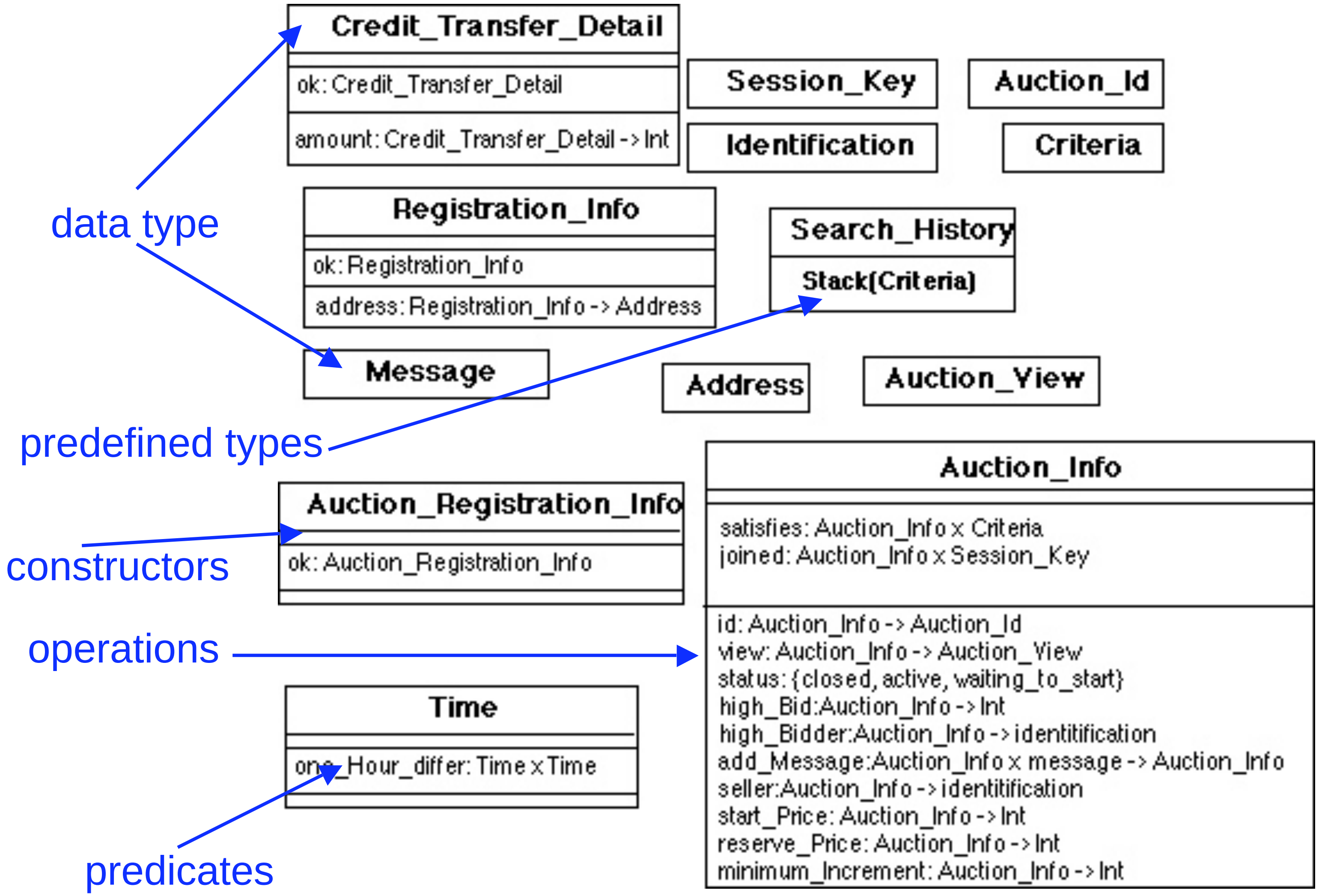
State observers

- simple dynamic system characterized by its states and labelled transitions
- labelled transition =
  state change + label (set of elementary interactions with external world)
- states abstractly characterized by “state observers”

Auction System: task 3 (cont.)

- Data View

Data used to type parameters and results of state observers and elementary interactions



Auction System: task 4

- find the **properties** about AuctionSystem by filling **“forms”** generated by the elementary interactions and state observers found in the previous task **systematically covering** “all” its aspects

  based on a many-sorted, first-order, CTL*-style temporal logic with edge formulae

- In the meantime
  - previous diagrams may be modified
  - new state observers may be added

  (thus the forms to be filled may be updated consequently)
  - original use case based requirement specification may be modified to reflect the better insights on the AuctionSystem gained while looking for properties

Auction System: task 5

(sample) Properties on CUSTOMER_JOIN_AUCTION

Form fragment

- pre/postcondition

  if CUSTOMER_JOIN_AUCTION(sk) **happen then**
  …condition about state observers on source state (of any transition having that elementary interaction in its label) …
  if CUSTOMER_JOIN_AUCTION(sk) **happen then**
  …condition about state observers target states (of any transition having that elementary interaction in its label) …

- Problems/Questions
  - Does the included use case search item ends having selected one auction or one item?
  - Can an auction selected by search item be in any status (e.g., closed or not yet started)?
  - Can a Customer try to join a closed or not-started auction?
  - Can a Customer join an auction to which (s)he is already joined?

(sample) Properties on CUSTOMER_JOIN_AUCTION

if CUSTOMER_JOIN_AUCTION(sk) **happen then**

exists id:Identification **s.t.** is_Identifed(id,sk) **and**

exists aid:Auction_Id **s.t.**

selected_Auctions(sk) = {aid} **and**

status(infoAbout(aid)) = active **and**

joined^{nxt}(sk,aid) **and**

in any case next

AS_SHOW_AUCTION(sk,view(infoAbout(aid))
happen

State observer on source state

State observer on target state

(sample) Properties on credit

Form fragment

- how change

if credit(id) = x **and** credit^{nxt}(id) = y **and** $x \neq y$ **then**

…condition about id, x and y and
some elementary interactions must happen in that transition (belong to its label) …

Property

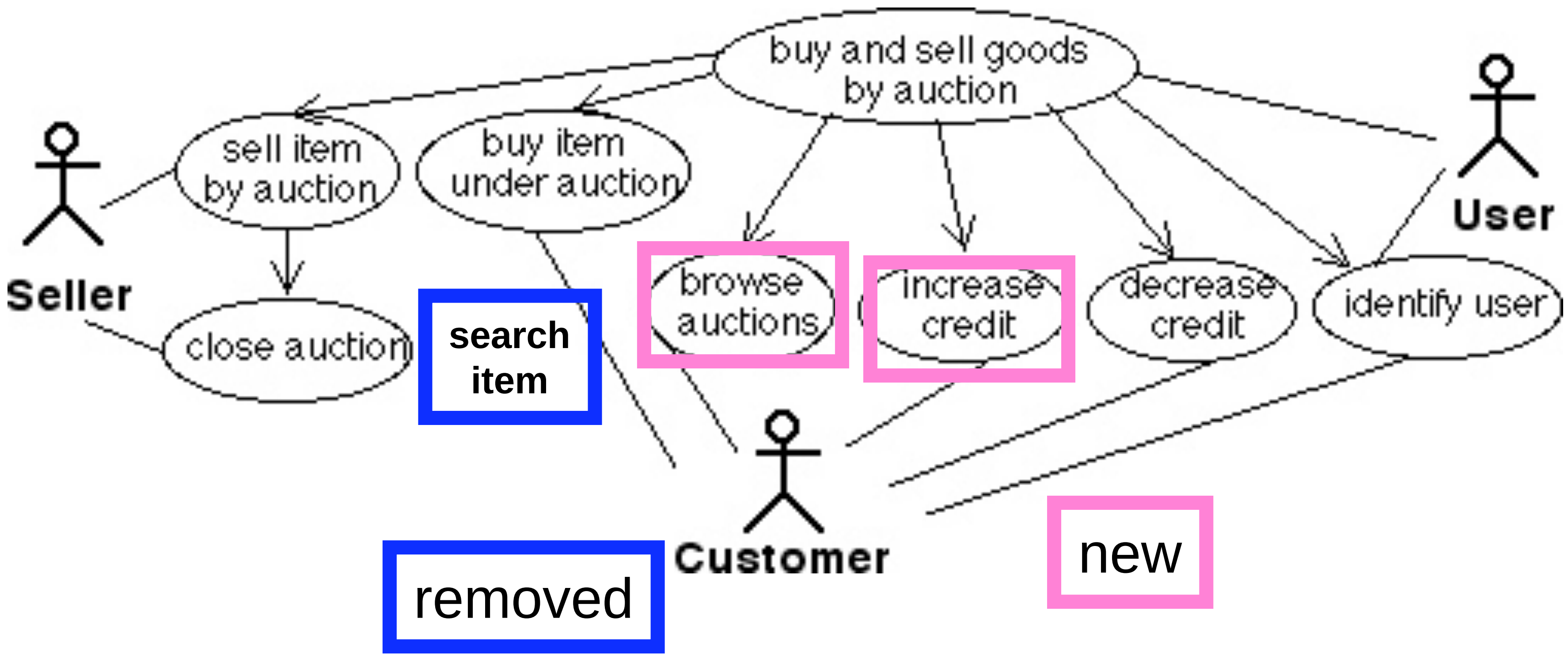
if credit^{nxt}(id) = credit(i) - i **and** i> 0 **then**
exists sk:SessionKey, ai:AuctionId **s.t.**
AS_BID_OK(sk,ai,i) **happen and** is_Identified(id,sk)

- Problems/Questions
  - It is true that a Customer using the AuctionSystem only for selling items will be never able to collect her/his money? Moreover, can a buying Customer recover her/his money when (s)he is no more interested in buying?

Auction System: task 5

Revised Use case based requirement specification

New Use case diagram



Revised “buy item under auction”

Intention in Context: The intention of the Customer is to follow the auction, ...

Primary Actor: Customer

Precondition: Customer already identified and selected one active auction NEW

Main Success Scenario:

1. Customer searches for an item under auction (search item). REMOVED
2. Customer requests to join the item auction.
3. System presents a view of the auction

...

buy item under auction (contd)

Steps 4-5 can be repeated according to the intentions and bidding policy of the Customer

4. Customer makes a bid on the item to System.
5. System validates the bid, records it, secures the bid amount from Customer's credit, … informs Participants of new high bid REMOVED, and updates the view of the auction
6. System closes the auction with a winning bid by Customer.

Extensions: ...

buy item under auction (extens.)

Extensions:

2a. C requests System not to pursue item further:

2a.1. System permits Customer to choose another auction, or go back to an earlier point in the selection process; uc continues at step 2.

3a The Customer is the Seller of the auction; System informs Customer that (s)he cannot join the auction. Use case ends with failure NEW

3a. System informs Customer that auction has not started: use case ends in failure. REMOVED

3b. System informs Customer that auction is closed: use case ends in failure. REMOVED

…

Conclusion

- proposed a method to review use case based requirements by building a companion Formally Grounded specification
  - as result
    - initial requirements examined in a systematic way by looking at the various aspects of the considered system
    - original use case based requirements updated whenever an aspect of the system is enlightened
    - the Formally Grounded specification (diagrams plus textual annotations) could be used as an alternative requirement document
    - the CASL-LTL specification corresponding to the Formally Grounded one is also available, e.g., for formal analysis
- building directly the Formally Grounded specification not as much as effective as the proposed combination
  - Formally Grounded specification ingredients (elementary interactions and state observers) finer grained than system functionalities, thus hard to find them just considering the problem

Auction System Experiment

- medium-size case study
- starting use case requirements
  - not produced by ourselves
  - quite accurate and presented using a well-organized template
- positive outcome
  - detected many problematic or unclear aspects in the original use case based Requirements
    - explicit auctions browsing functionality
    - auctions should be performed in a chat-like way
    - need for a decrease-credit functionality
    - two different Customers may be the same person
    - a Customer may disconnect by the System by hers\his own choice, and not only after sometime (s)he is doing nothing
    - a Customer cannot unregister from the System when (s)he is the seller or has the high bid in an auction
    - made explicit that when a Customer unregisters any left credit is seized by the Auction System owner
    - …

Related work: inspection techniques

- Inspection techniques for requirement spec:
  ad hoc techniques or check-lists

  ‘‘Is there any missing functionality, that is, do the actors have goals that must be fulfilled, but that have not been described in use cases?’’

- Our ‘‘inspection’’: build a companion formal specification with a form-filling technique
  leads to a systematic and precise requirement examination

  ‘‘find and list all the ways the *credit* state observer may be updated in the various scenarios of all use case’’

  -> credit decreasing needed !!

Our high quality requirements method

Task 1: use case diagram &
descriptions (Sendall & Strohmeier)

Iterative construction of the specification:

Task 2: initial Context View
configuration diagram & cooperation diagram

Task 3: for each use case
- *elementary interactions & state observers*
  –> cooperation diagram (update)
- *Data View* (data structures)

Task 4: properties (form filling method)
–> update elem inter, state obs, data struct

Task 5: in parallel, record questions
–> update use case accordingly

and more …

- General requirement formal specification development method

- Initially aimed for CASL/CASL-LTL languages

- Could be used with other specification languages

(colored/high level Petri nets, …)

- May be used in combination with informal notations/methods: use cases, UML, problem frames, …

- Architectural styles may be used to work further

towards the design specification

Complementary related works

- How to write readable CASL specifications, avoiding semantic pitfalls http://www.brics.dk/Projects/CoFI
  - Roggenbach and Mossakowski for the basic data types library
  - Bidoit and Mosses in the CASL reference manual
- Bidoit and Hennicker [e.g. FOSSACS02] explore the use of observability concepts which are found to be useful and relevant for writing specifications, and the combined use of constructors and observers
- Blanc [PhD 2002, Cachan] proposes guidelines for the iterative and incremental development of specifications
- Choppy and Reggio [WADT99] propose to help requirement analysis by generating CASL and CASL-LTL skeletons associated with Jackson's problem frames (used as structuring concepts to start the problem analysis)
- Choppy and Heisel [WADT02] propose to go on with using the structuring concepts provided by architectural styles to support design specifications and explore the combination with the problem frames used to begin with

Related work

- formal specification of requirements, e.g.
  - A. van Lamsweerde and his group
    - formally specifications of goal-oriented requirements plus analysis by means of formal techniques
  - R.Dromey
    - “Behaviour Tree” a formal-visual notation to specify the requirements, and a method to derive from them the architectural structuring of the system

Different aims

- “more precise” specification of requirements, e.g.
  - S. Sendall and A. Strohmeier
    - operation schemas (written in OCL) and system interface protocols (UML statecharts) to complement use cases
  - E. Astesiano- G. Reggio
    - Tight-structured UML based method for the precise specification of the requirements, where use case are modelled by statecharts

No validation \ inspection method