UNIVERSITY OF
LEICESTER

MGS 2022

GT4SE
Graph Transformation
for Software Engineers

Graph Transformation: Foundations and Applications in Software Engineering

Reiko Heckel

University of Leicester, UK

Based on book with Gabriele Taentzer

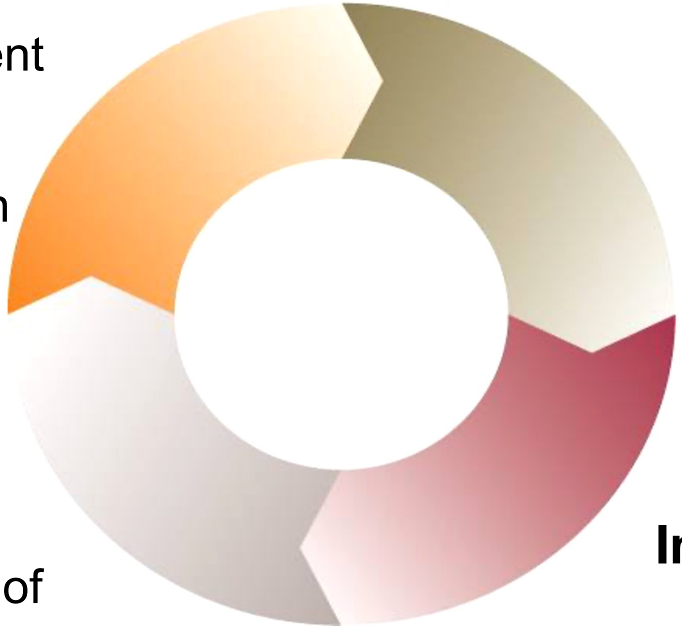
Philipps Universität Marburg, Germany

Graph Transformation-Based Software Engineering

Requirements analysis

5: Detecting inconsistent requirements

6: Service specification and matching



Testing and Analysis

7: Model-based testing

8: Reverse engineering

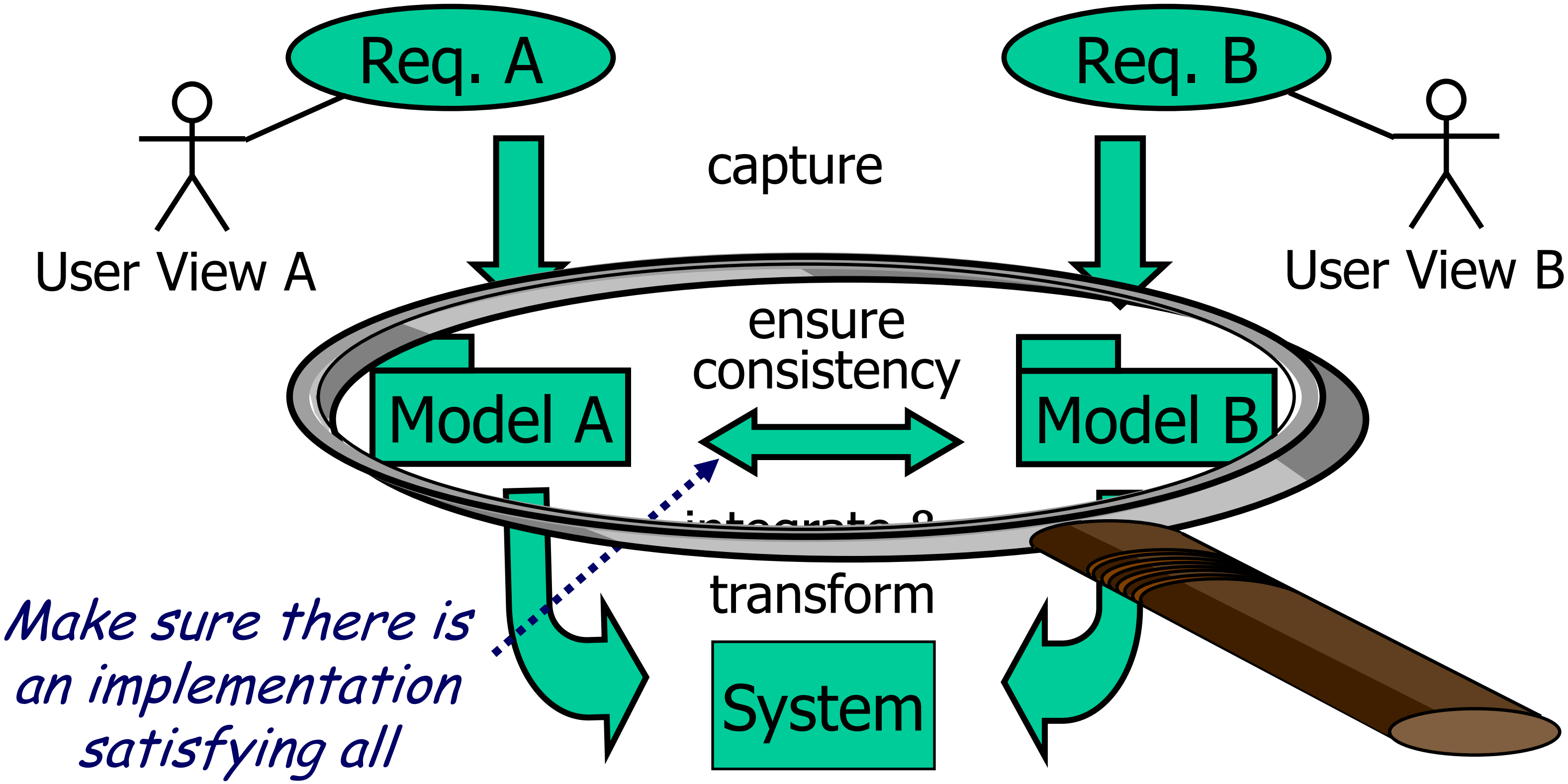
Software design

9: Stochastic analysis of dynamic architectures

Implementation

Detecting Inconsistent Requirements

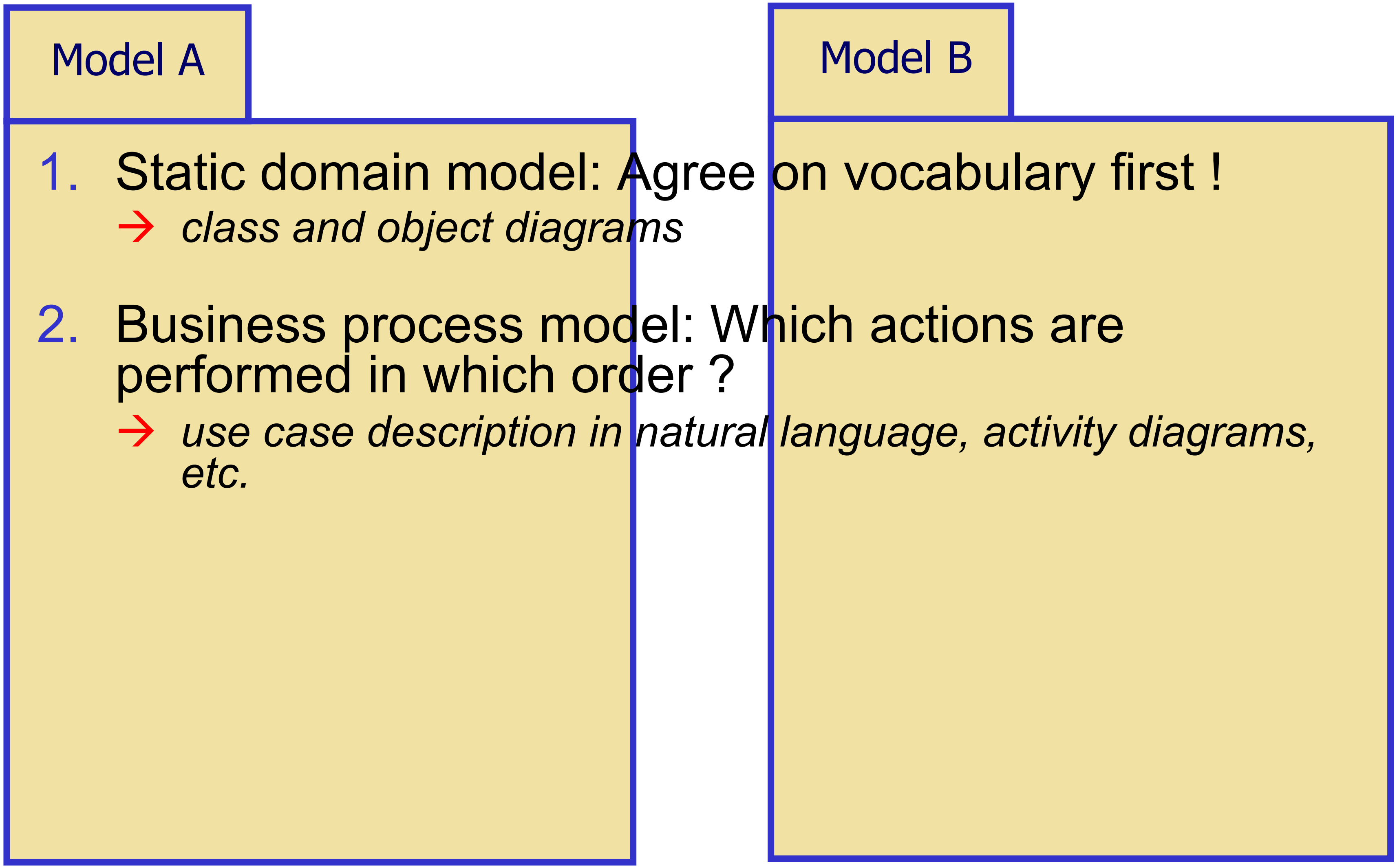
Integration of Views



Make sure there is an implementation satisfying all requirements !

1. Aspects of requirements models
2. Conflicts between functional requirements
3. Theory and tool support

Aspects of Requirements Models



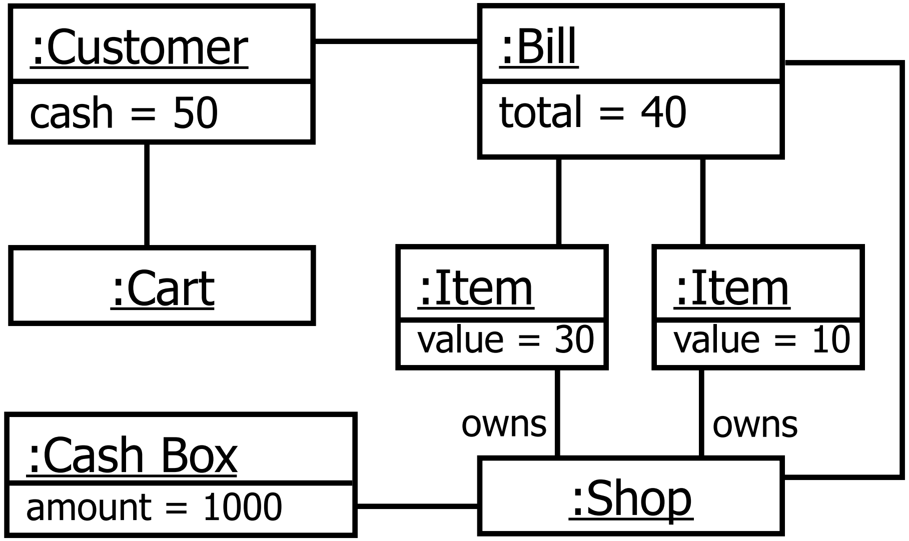
1. Static domain model: Agree on vocabulary first !
   → *class and object diagrams*

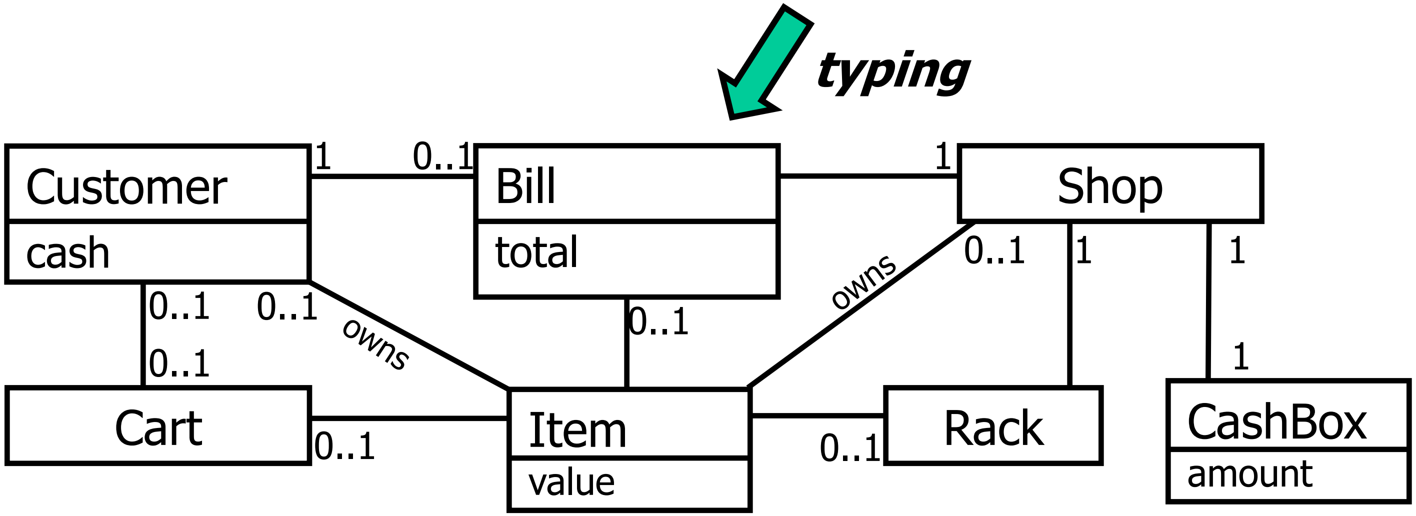
2. Business process model: Which actions are performed in which order ?
   → *use case description in natural language, activity diagrams, etc.*

Structure: Class and Object Diagrams

✓ formal, e.g., attributed graphs at the type and instance level

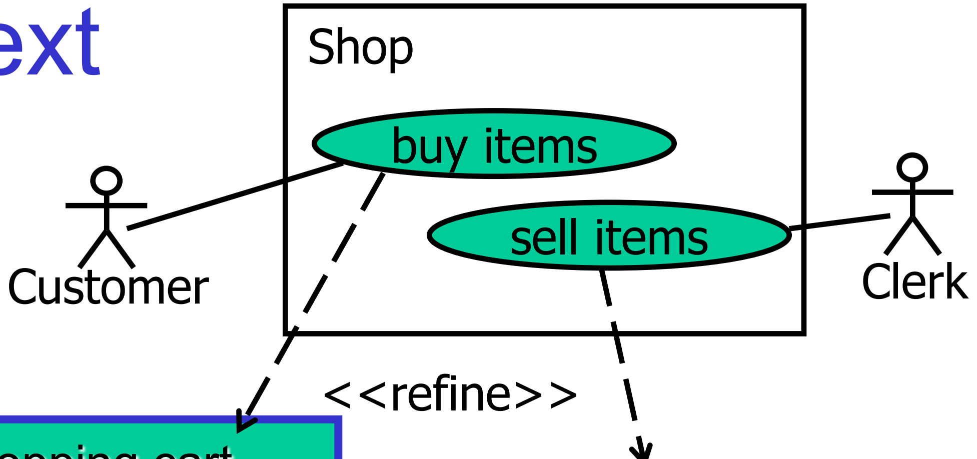
✓ established techniques for view integration





Behaviour: Use Cases as Structured Text

✓ based on vocabulary of integrated domain model



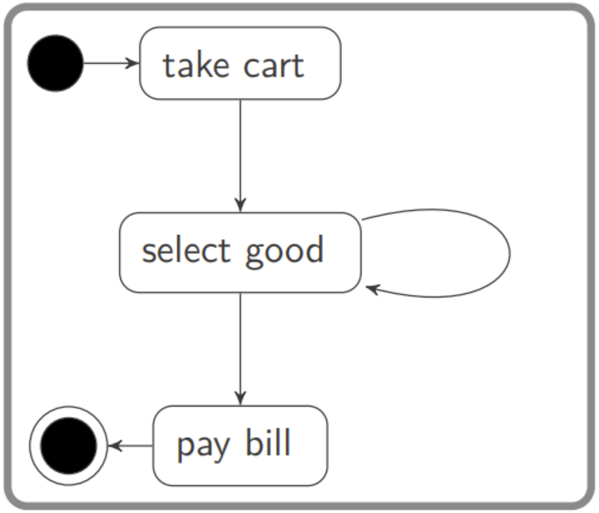
- take shopping cart
- select items from rack
- take items out of cart
- pay required amount
- collect items

- create empty bill for new customer
- take items out of customer's cart
- add them to the bill
- collect payment
- pack and give items to customer

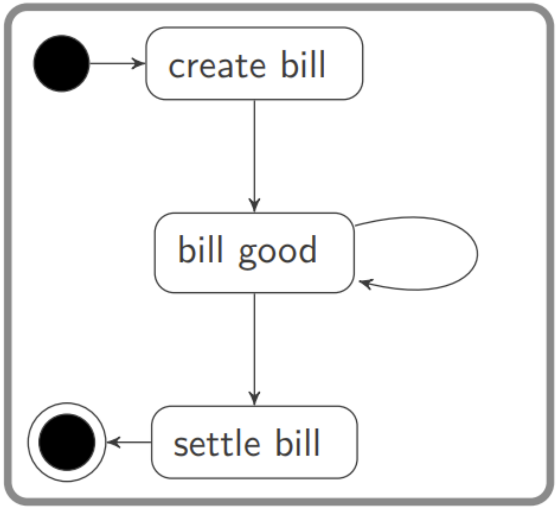
✗ no way to tell if views are consistent

Behaviour: Refinement by Activity Diagrams

Buy goods:



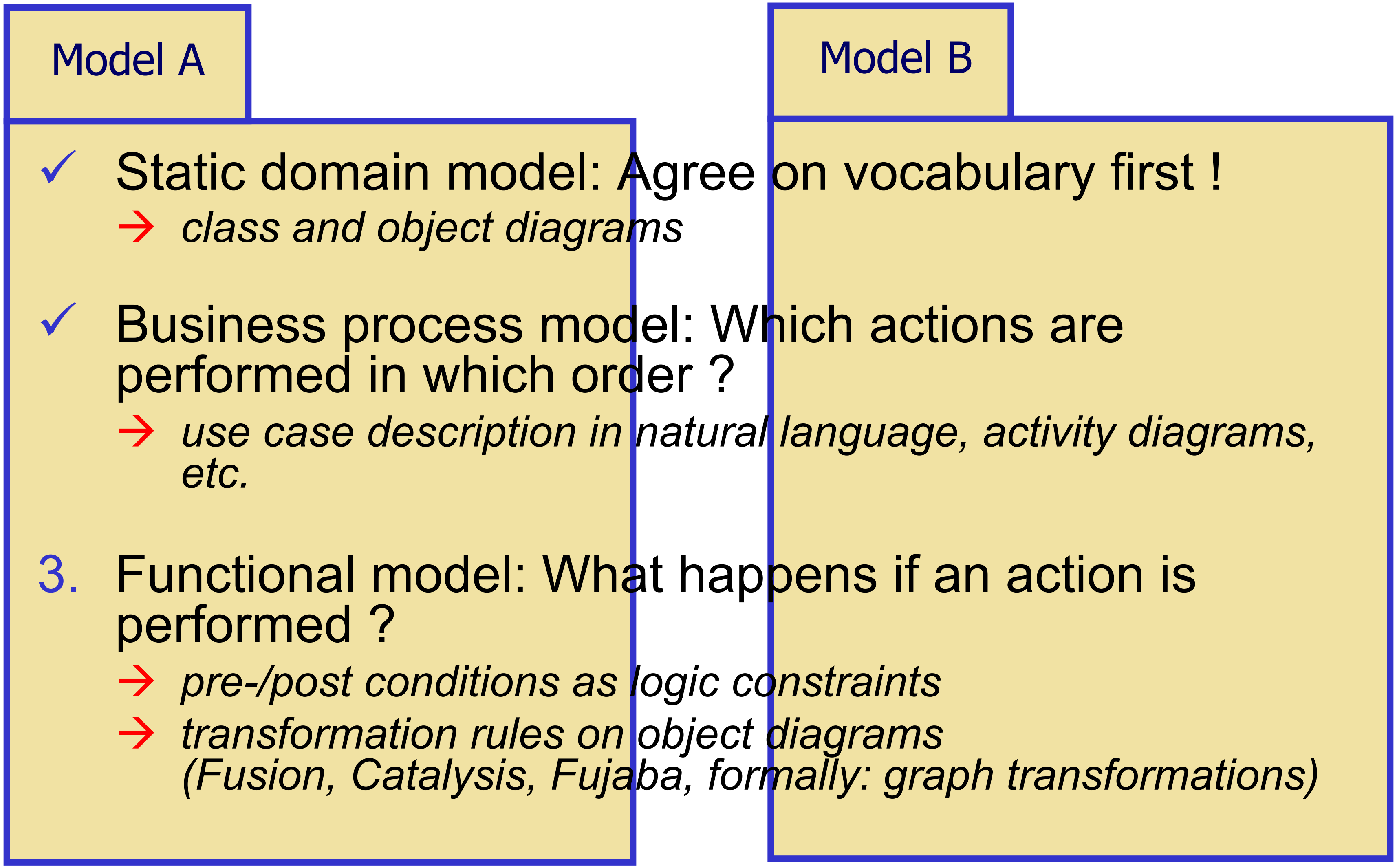
Sell goods:



[HT20]

- Are they consistent with the class model?
- Are the processes consistent with each other?
- Are there conflicts between then basic actions?

Aspects of Requirements Models



✓ Static domain model: Agree on vocabulary first !

→ *class and object diagrams*

✓ Business process model: Which actions are performed in which order ?

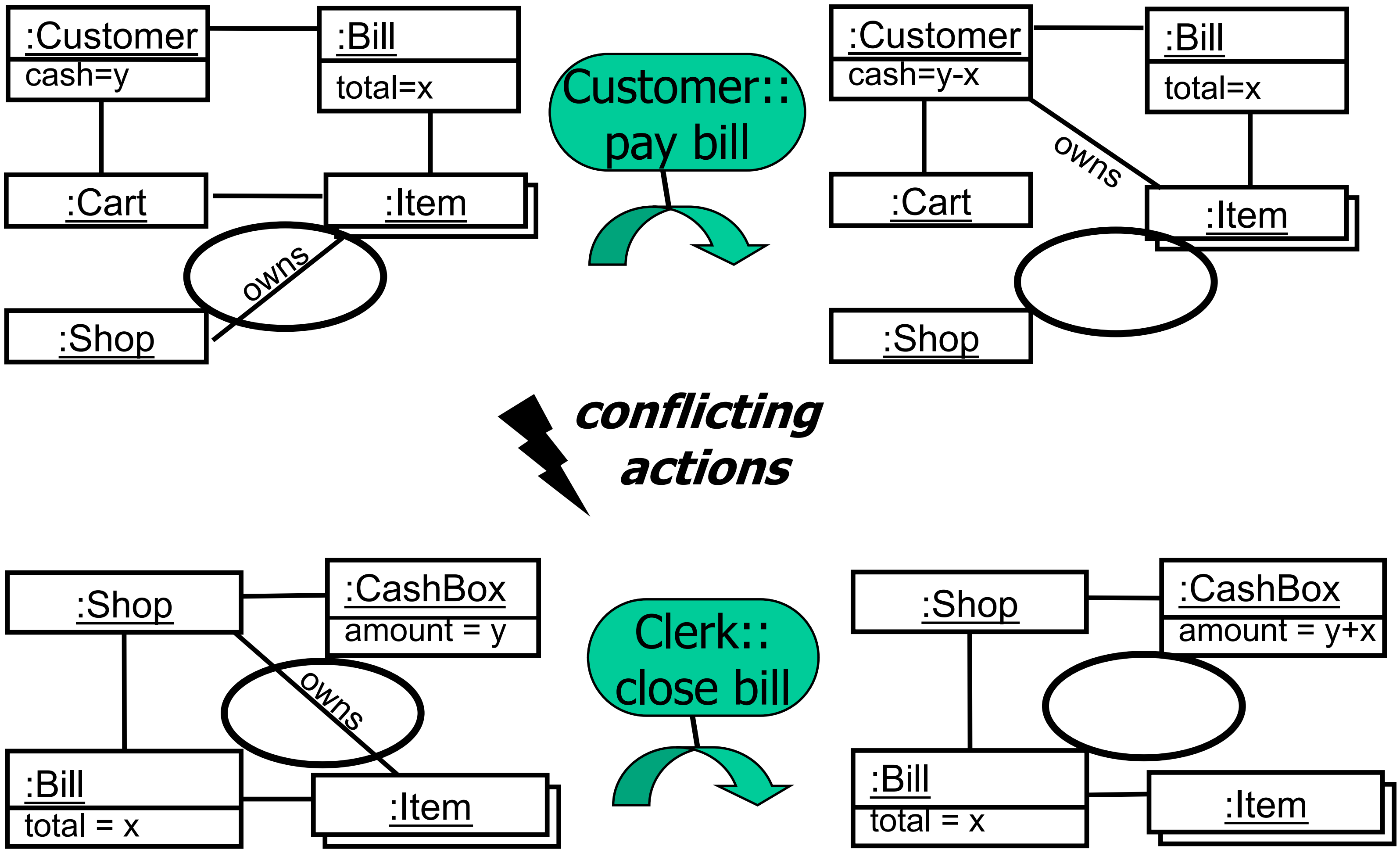
→ *use case description in natural language, activity diagrams, etc.*

3. Functional model: What happens if an action is performed ?

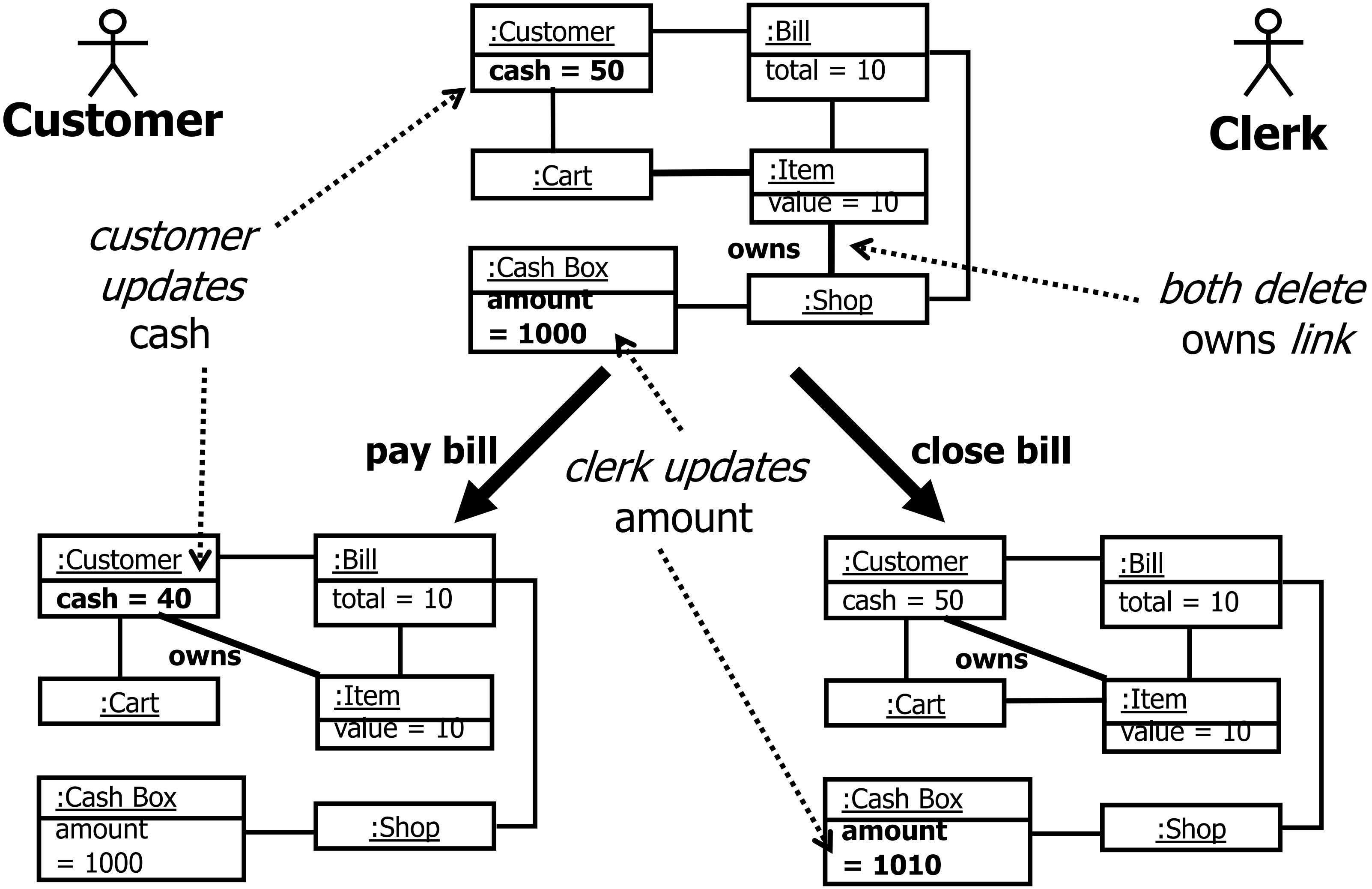
→ *pre-/post conditions as logic constraints*

→ *transformation rules on object diagrams (Fusion, Catalysis, Fujaba, formally: graph transformations)*

Function: Rules on Object Structures



Conflicting Functional Requirements



customer updates cash

clerk updates amount

both delete owns *link*

Theory: Independence, Causality and Conflicts in Graph Transformation

- Alternative steps are *parallel independent* if they do not disable each other.

  Otherwise they are *in conflict*.

- Consecutive steps are *sequentially independent* if they may be swapped without affecting the result.

  Otherwise they are *causally dependent.*

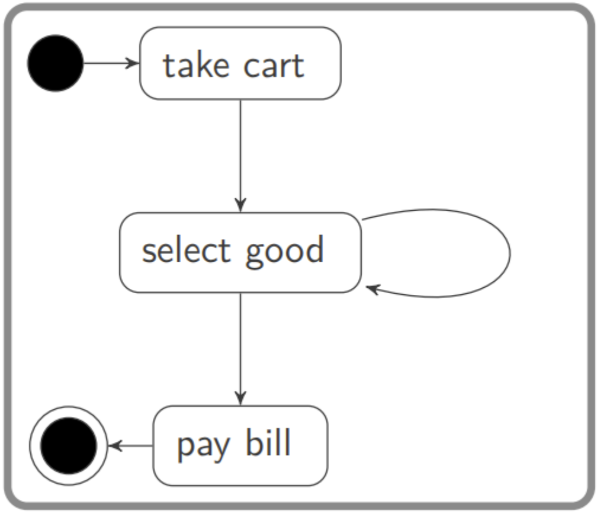
Aim: Find *potential* conflicts and dependencies between rules by **critical pair analysis**

$$\begin{tikzcd}
 & G \arrow[dl, "p_1"'] \arrow[dr, "p_2"] & \\
H_1 \arrow[dr, "p_2"'] & & H_2 \arrow[dl, "p_1"] \\
 & X &
\end{tikzcd}$$

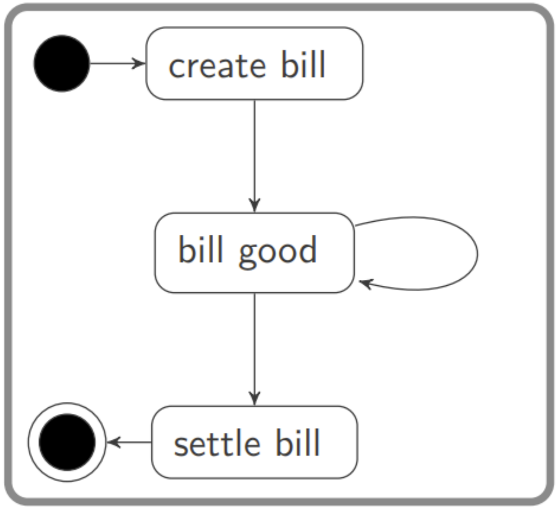
Characterization [EPS73]:
Two (alternative or consecutive) steps are independent **iff** all commonly accessed items are in read-access only.

Are these in conflict / dependent?

Buy goods:



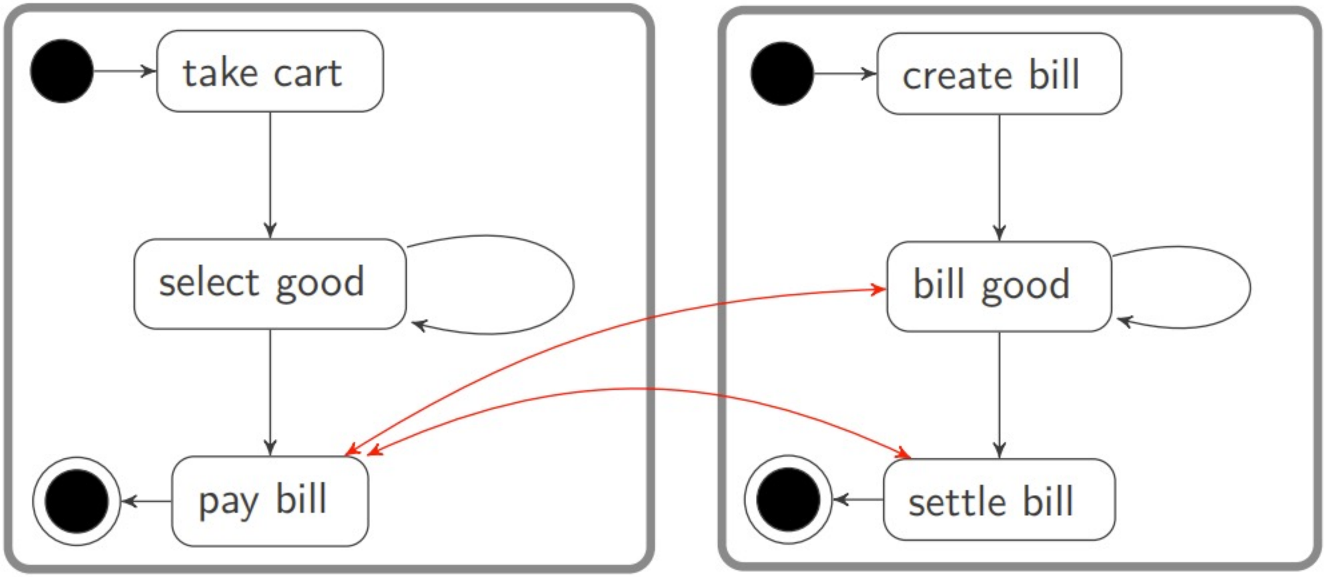
Sell goods:



[HT20]

- What conflicts and dependencies can arise between their activities?

Are these in conflict / dependent?
Potential conflicts

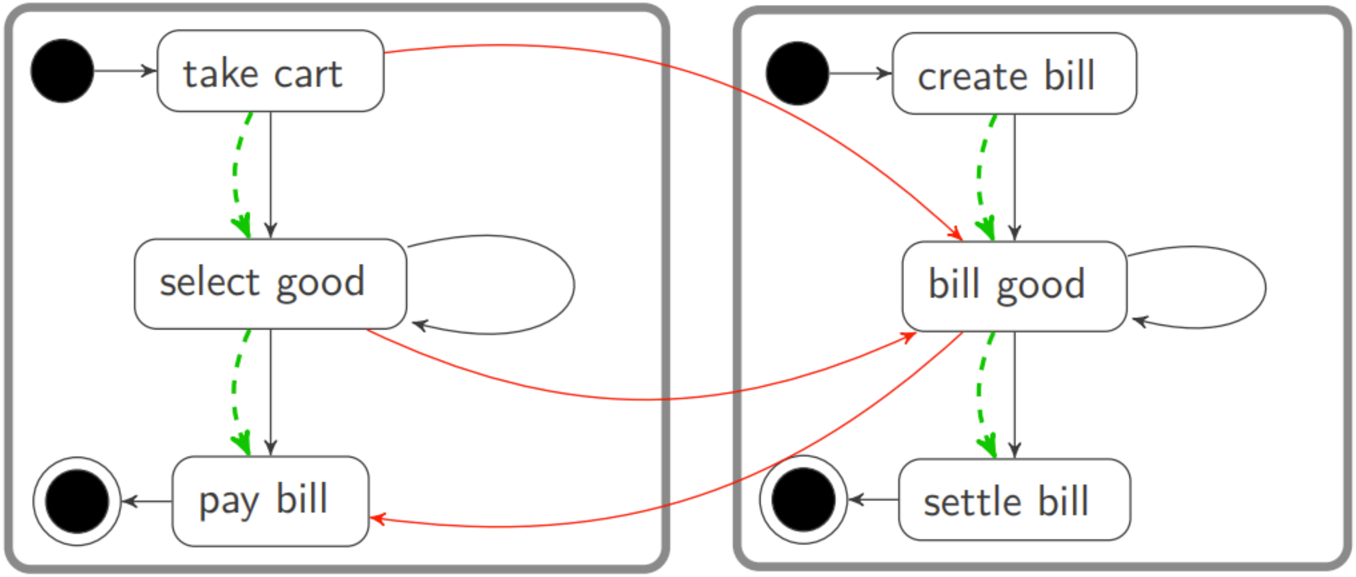


[HT20]

- What potential conflicts can arise?
- Can these be resolved by changes in the activity diagrams?

Are these in conflict / dependent?

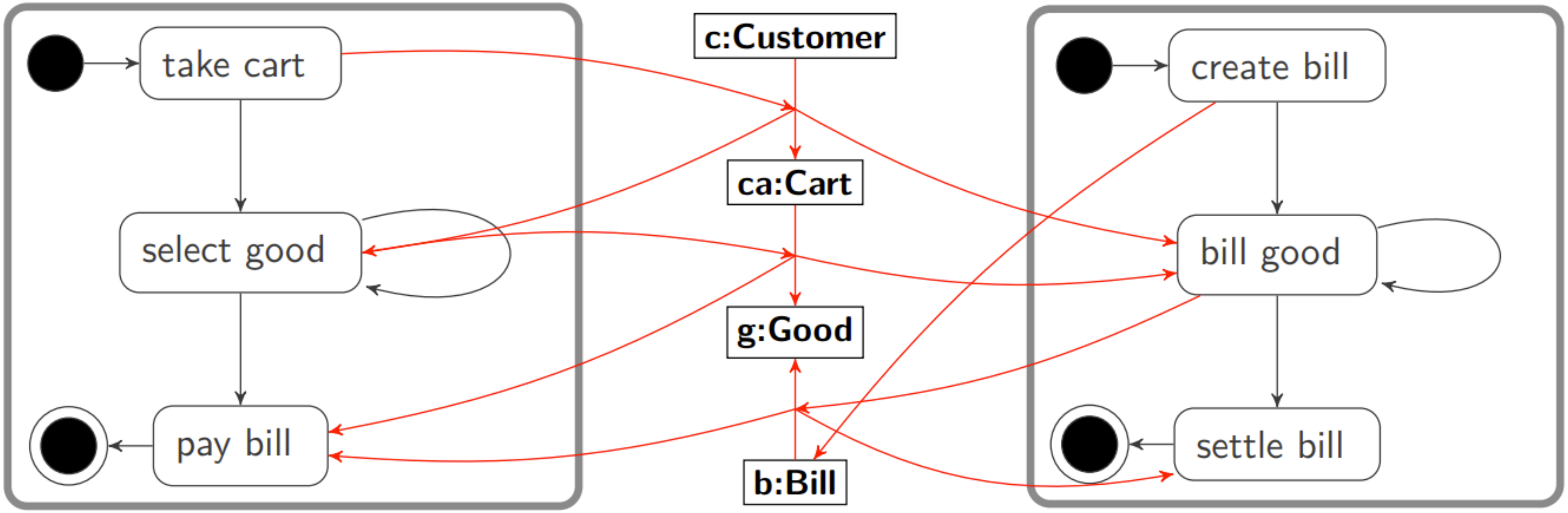
Potential dependencies



[HT20]

- Favourable: dependencies along the control flow
- Critical: dependencies between use cases of different views

Activity Diagrams with Dependency Reasons



[HT20]

- Objects and links to explain potential dependencies.
- Analogous to activity diagrams with object flow.

Formalise this, …

Transformations in conflict or dependent

➔ *Alternative or delayed matches*

Rules potentially in conflicts or dependent

➔ *Critical pairs*

Summary

- Requirements:
  - *Structure: Class diagrams*

    →Type graphs
  - *Function: pre- and postconditions*

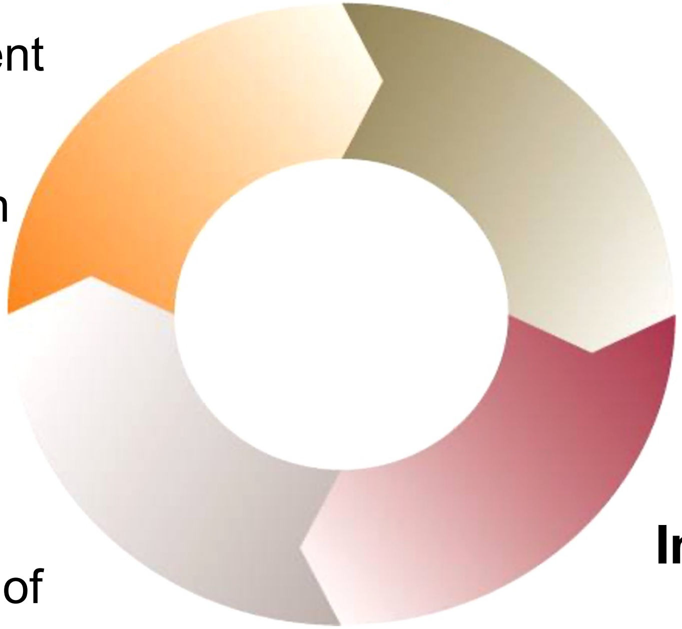
    →Rules
  - *Behavour: activity diagrams*

    →Control structures
- Consistency
  - *Structure vs function*

    →Typed graph transformation
  - *Function vs behaviour, between views*

    →Conflict and dependency analysis

Graph Transformation-Based Software Engineering



Requirements analysis

5: Detecting inconsistent requirements

6: Service specification and matching

Testing and Analysis

7: Model-based testing

8: Reverse engineering

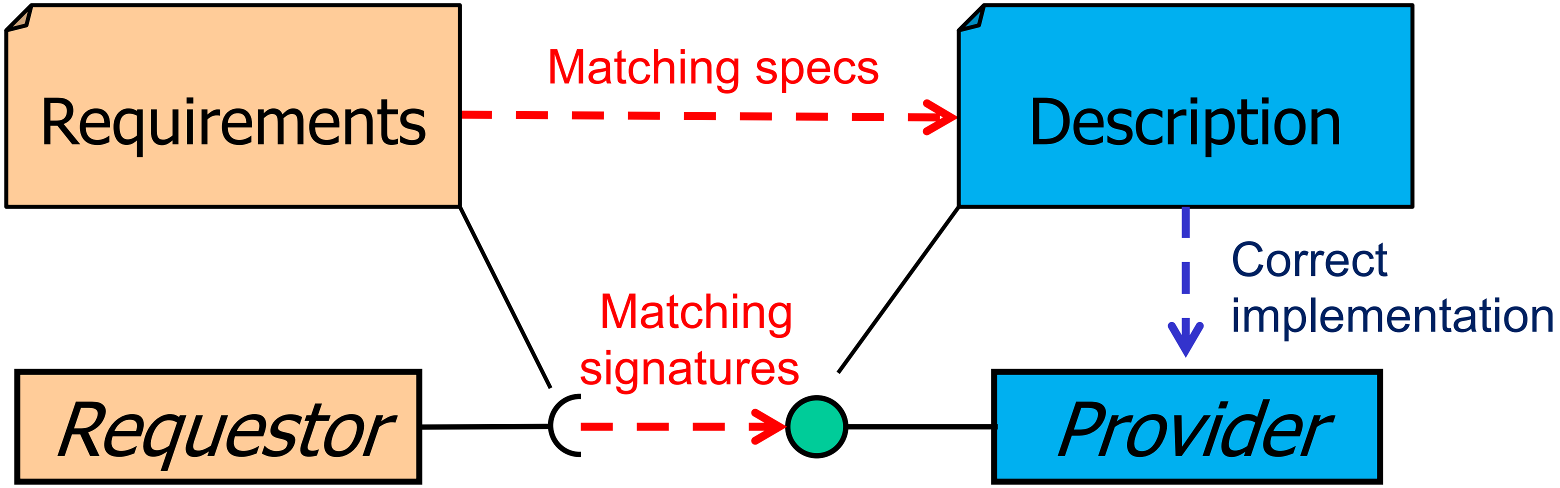
Software design

9: Stochastic analysis of dynamic architectures

Implementation

Service Specification and Matching

Consistency in Service-oriented Systems



External: between required and provided specifications

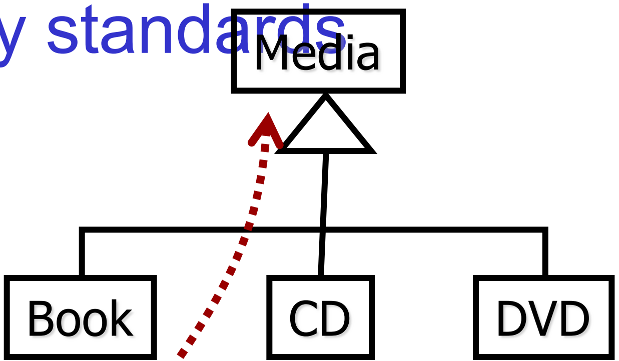
Matching data models and operations

Internal: between specification and implementation

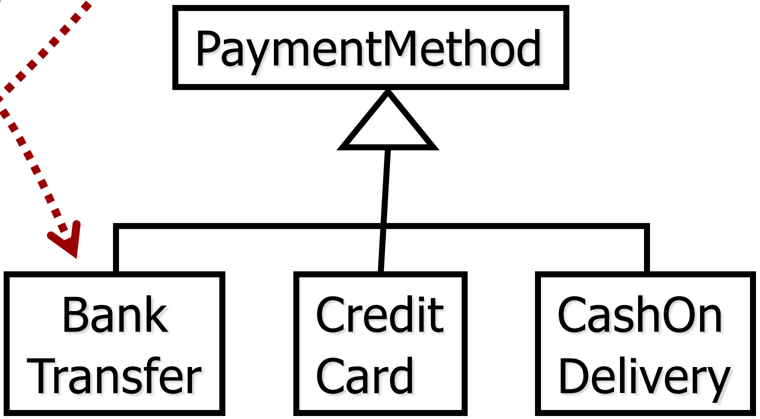
Testing and reverse engineering

Ontology:
Domain-oriented industry standards

Matching *requirements* to *descriptions*
requires a common understanding
of underlying concepts:



Requestor's requirement:
"I need an online *book* shop that
accepts payment by *bank transfer*."



Provider's service description:
"We sell all kinds of *media*.
You may pay via *credit card* or
bank account."

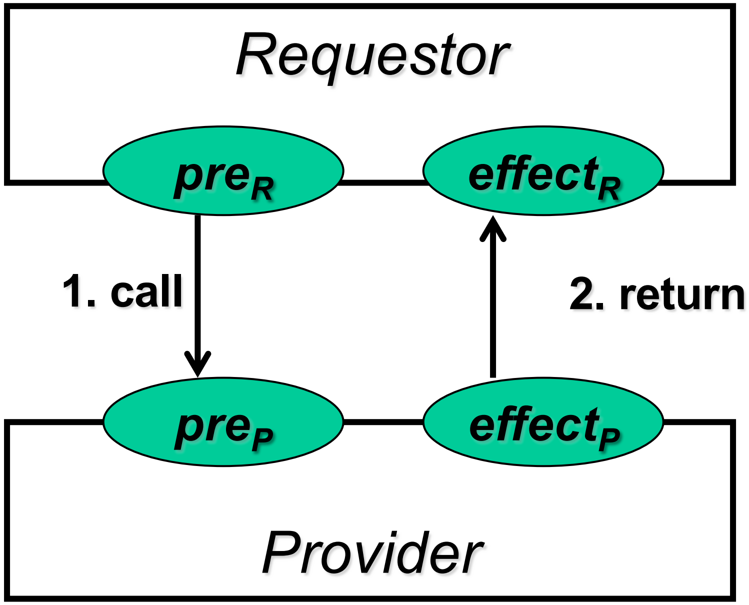
Design by Contract (Meyer, 1988)

- Interface is contract between requestor and provider
- Both expect benefits and accept obligations

contracts for ***payBill***	**obligations**	**benefits**
Requestor *Client* requirements	I provide account data.	I expect that the Bill will change status to „payed".
Provider *Shop* description	I guarantee that the Bill will change to "payed", you will get an ack, and I store your data.	You provide account data of the client who pays.

- Expressible in logic, behavioral models, OCL, etc.
- Here: visual contracts as visual preconditions and effects

Matching Requestor with Provider Pre- and Postconditions



Requestor guarantees $\boldsymbol{pre_R}$

→ Provider assumes $\boldsymbol{pre_P}$

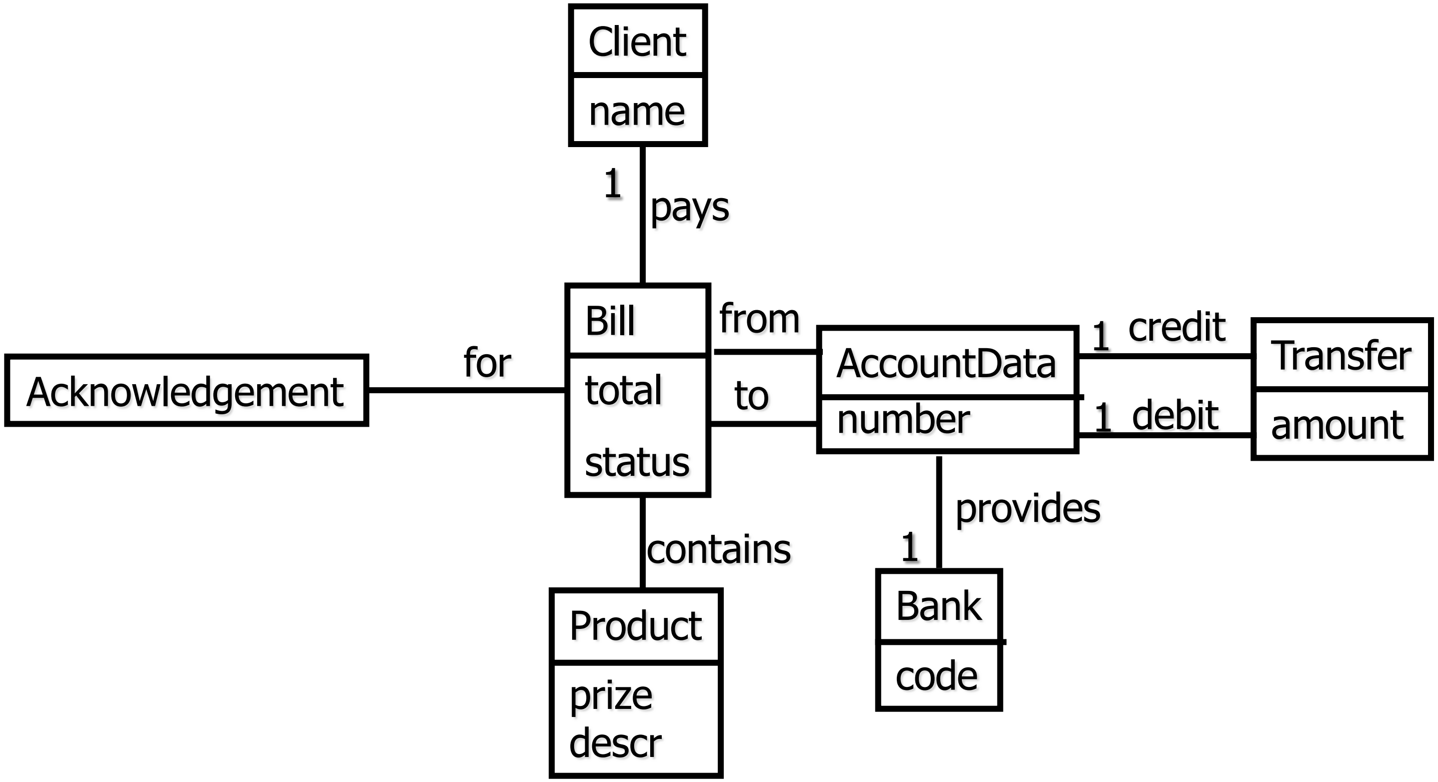
Provider guarantees $\boldsymbol{effect_P}$

→ Requestor assumes $\boldsymbol{effect_R}$

Requires

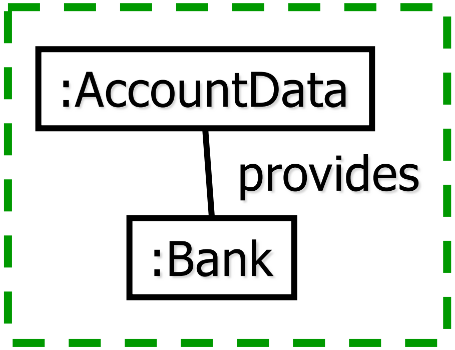
- conversion between data models
- or shared data model (ontology)

Shared Data Model (Ontology)

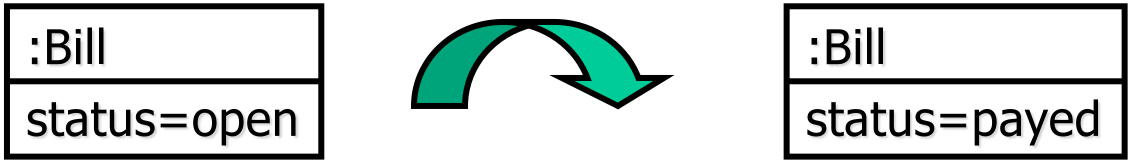


Requestor‘s Requirement: An Inquiry for a Contract

„I want to pay via bank account!“



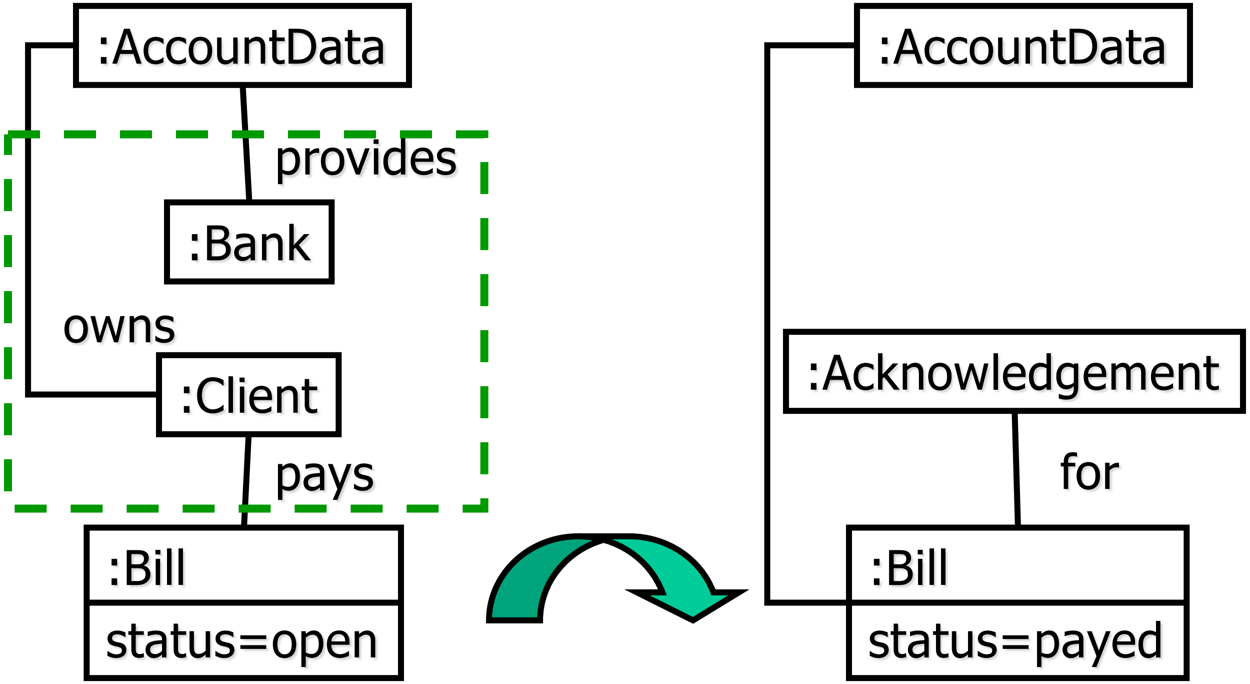
Pre: I provide account data (unchanged)



Effect:
I expect that the Bill will change status to „payed‟ (a transformation)

Provider‘s Description: A Contract Offer

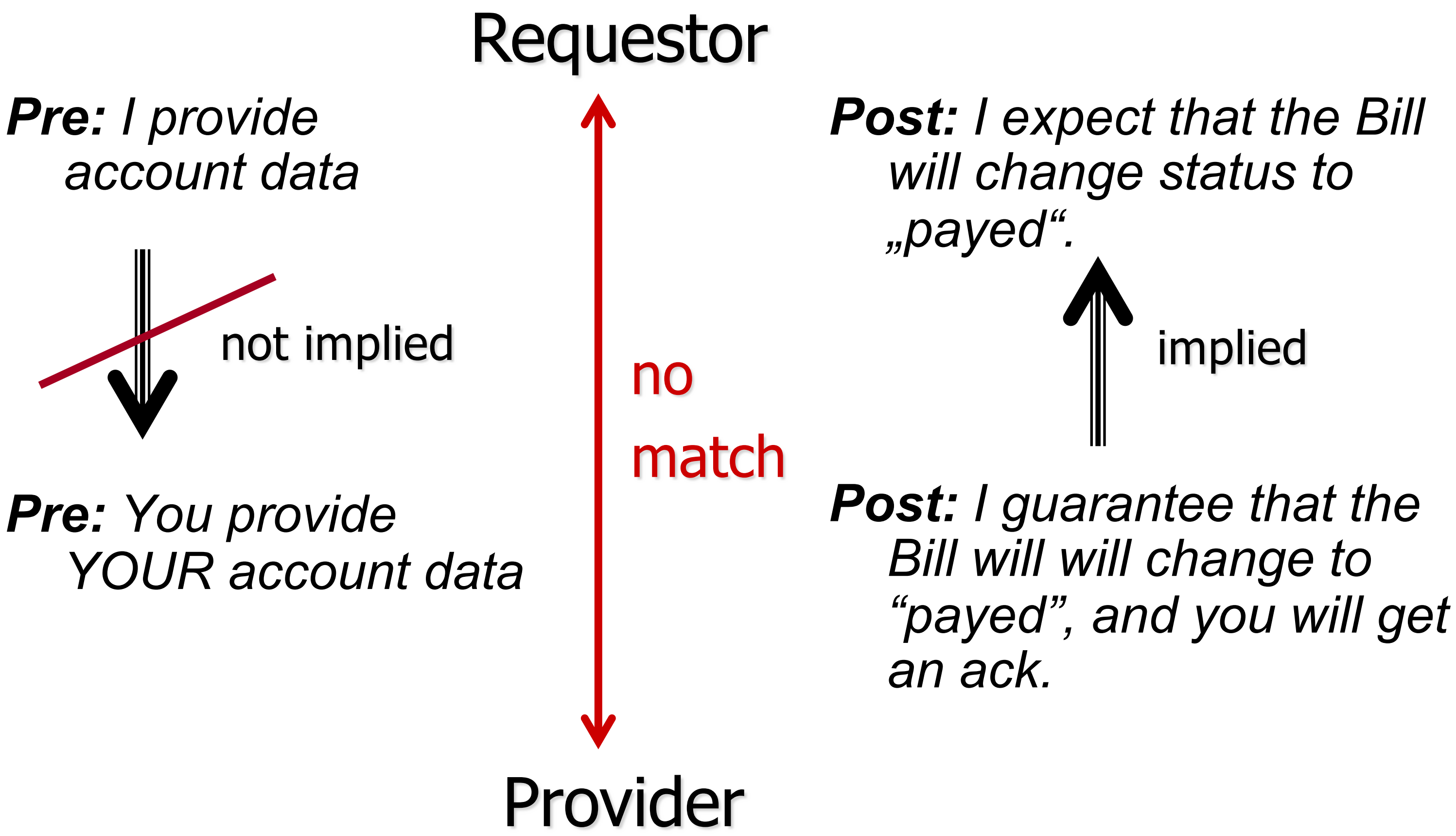
„You may pay via bank transfer!“



Pre: You provide account data of the client who pays.

Effect:
I guarantee that the Bill will change to "payed", you will get an ack, and I store your data.

Matching Inquiry and Offer



Requestor

Pre: *I provide account data*

not implied

Pre: *You provide YOUR account data*

no match

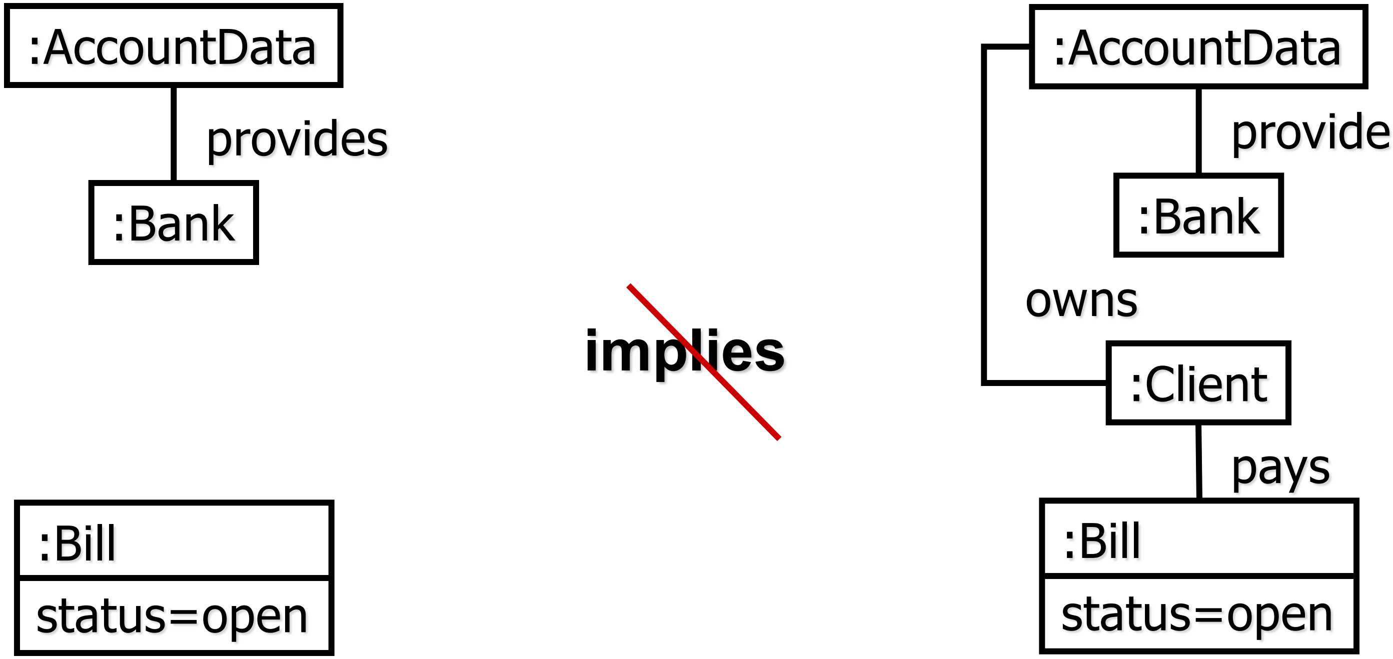
Post: *I expect that the Bill will change status to „payed“.*

implied

Post: *I guarantee that the Bill will will change to “payed”, and you will get an ack.*

Provider

Inquiry and Offer: Preconditions

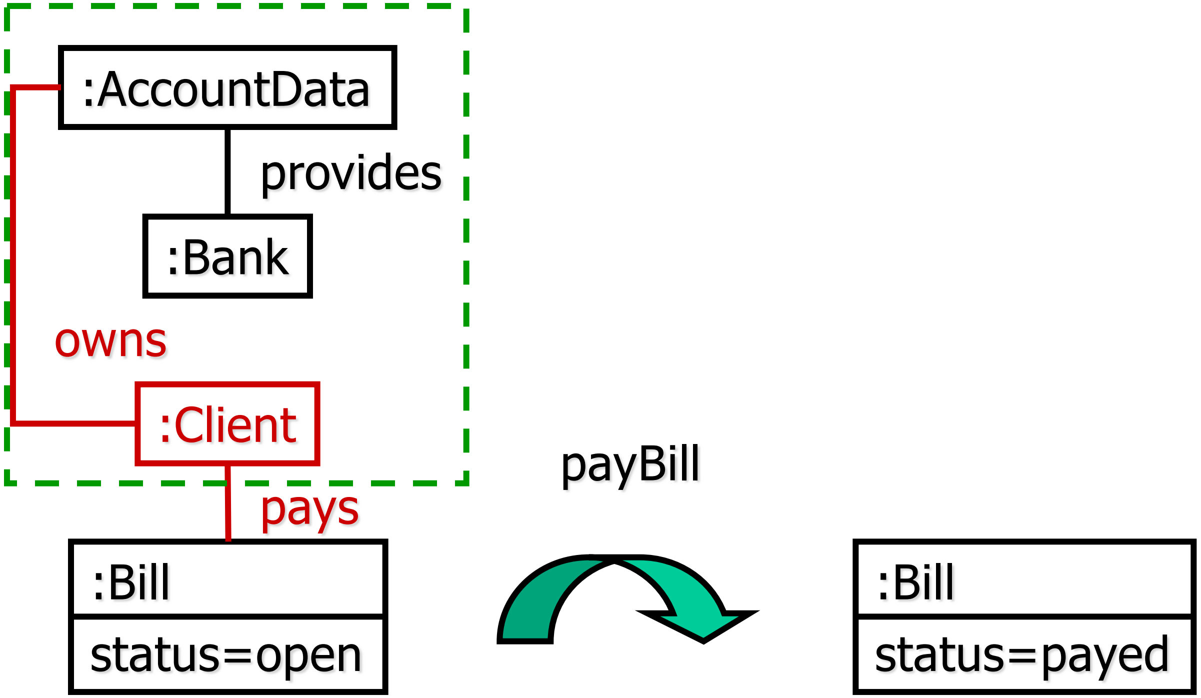


Pre_{Req} implies Pre_{Pro} **iff** Pre_{Pro} **can be embedded in** Pre_{Req}

„everything assumed by provider is guaranteed by requestor“

Requestor‘s service requirement: Extended precondition

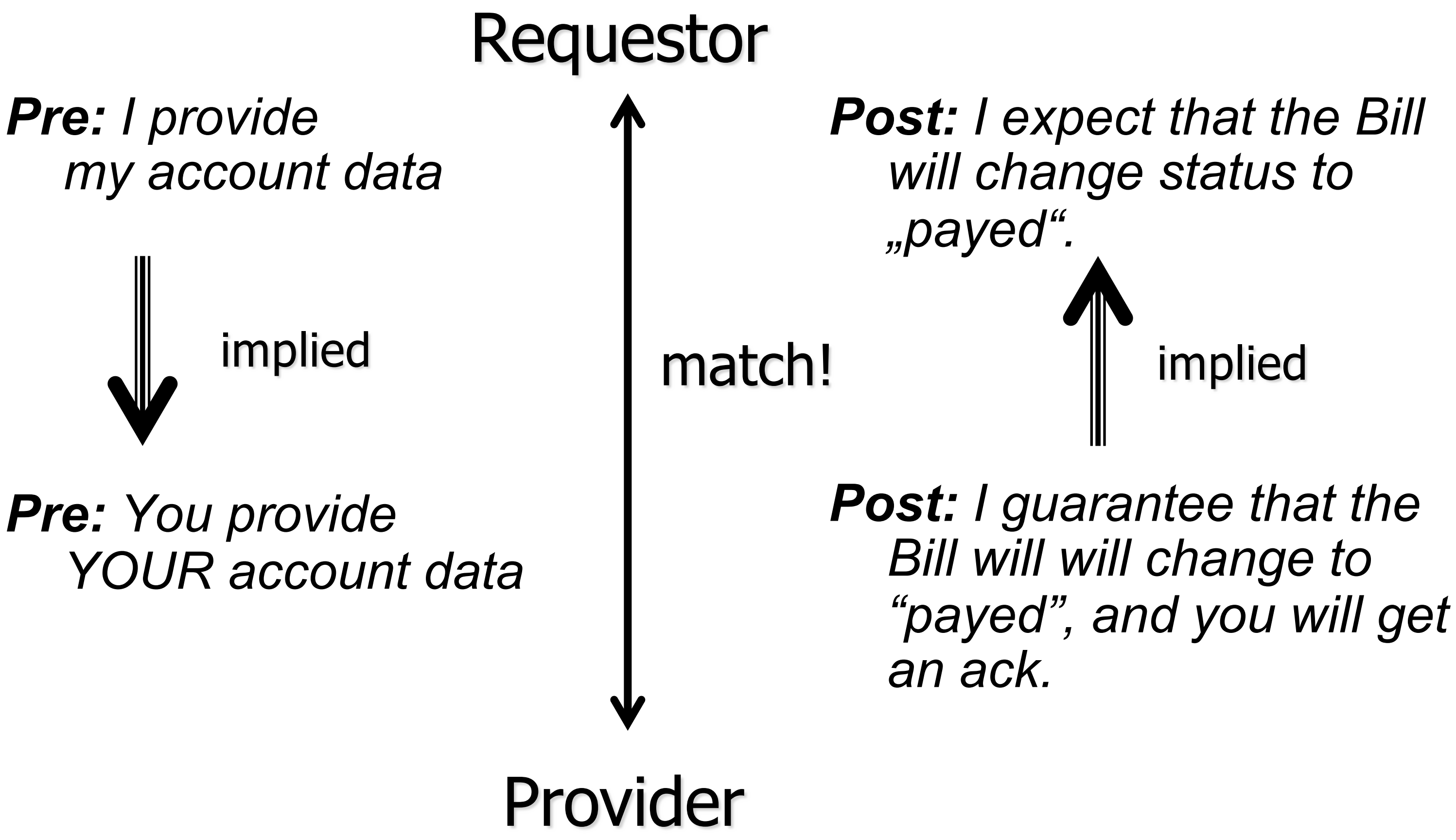
„I want to pay via bank transfer!“



Pre: I provide account data *of the client who pays.*

Post:
I expect that the Bill will change status to „payed“

Matching Inquiry and Offer



Requestor

Pre: *I provide my account data*

implied

Pre: *You provide YOUR account data*

match!

Post: *I expect that the Bill will change status to „payed“.*

implied

Post: *I guarantee that the Bill will will change to “payed”, and you will get an ack.*

Provider

Formalise this, …

Transformations in conflict or dependent

- ➔ *Alternative or delayed matches*

Rules potentially in conflicts or dependent

- ➔ *Critical pairs*

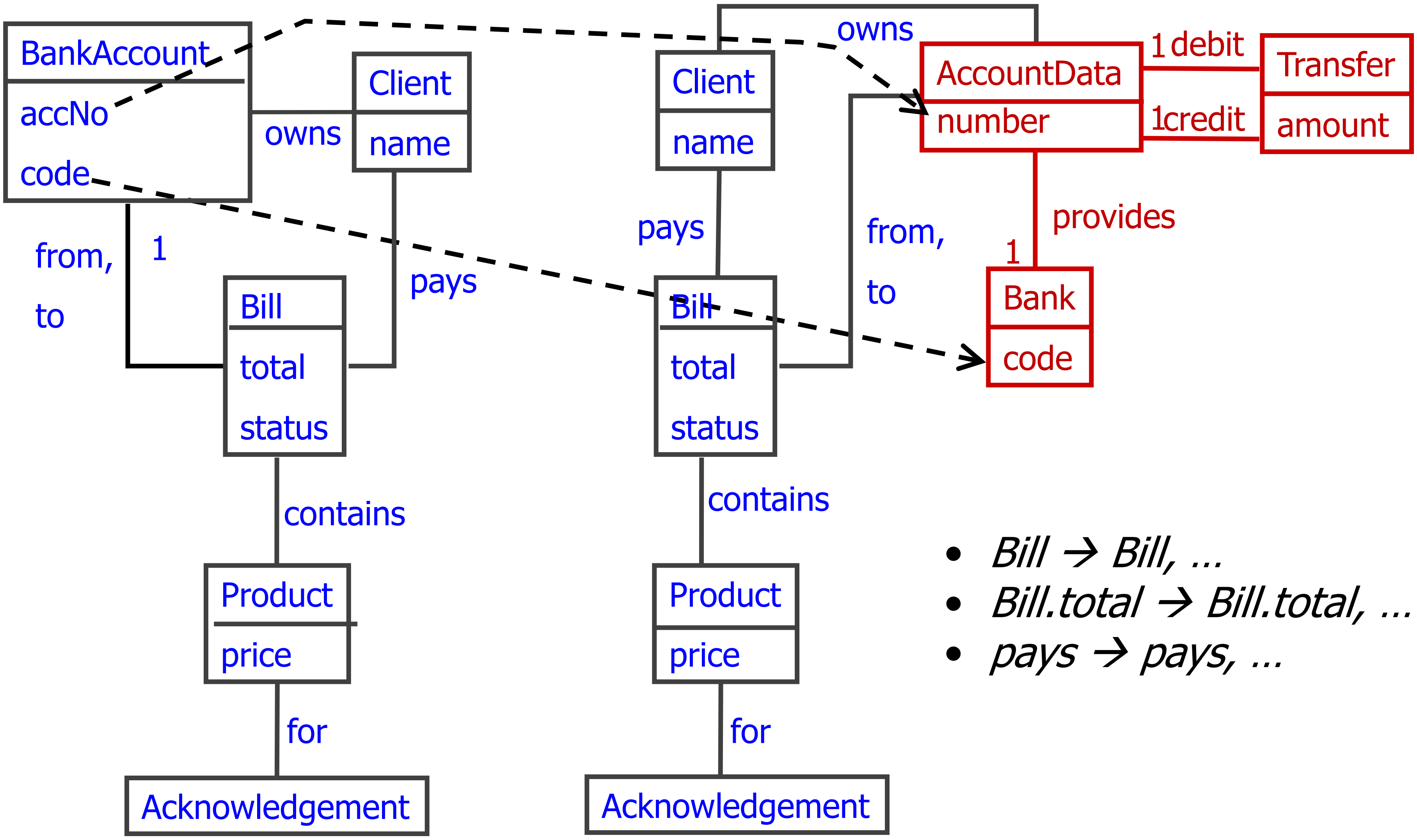
Service specs over local data models

- ➔ *Mapping between data models*
- ➔ *Translate state graphs and rules*

Visual contract = precondition + effect

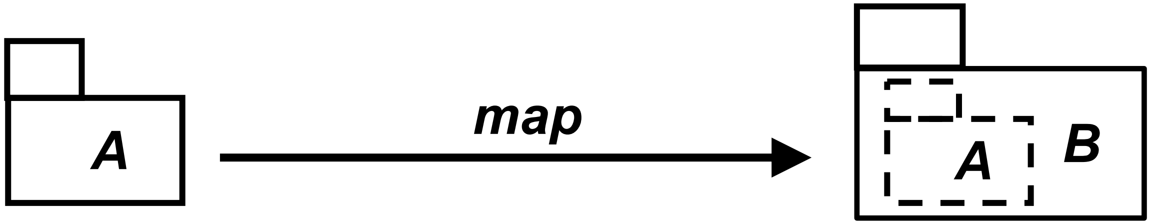
- ➔ *Separate effect as minimal rule*
- ➔ *Compare preconditions*

Data Models: Shop → Agent



- *Bill → Bill, …*
- *Bill.total → Bill.total, …*
- *pays → pays, …*

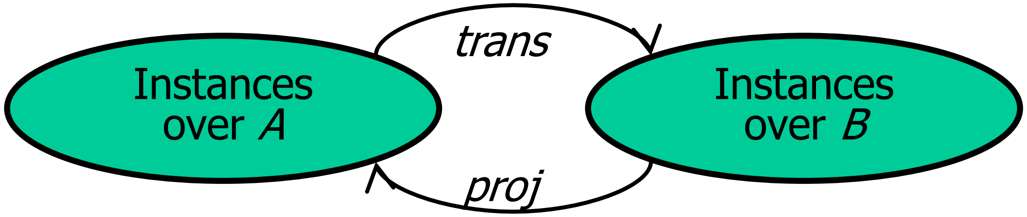
Mappings Between Data Models



Data models:

- *covariant* translation of instances of ***A*** into instance of ***B*** *without loss of data*
- *contravariant* projection of instances of ***B*** to instances of ***A*** *losing all data typed over **B** – **A***

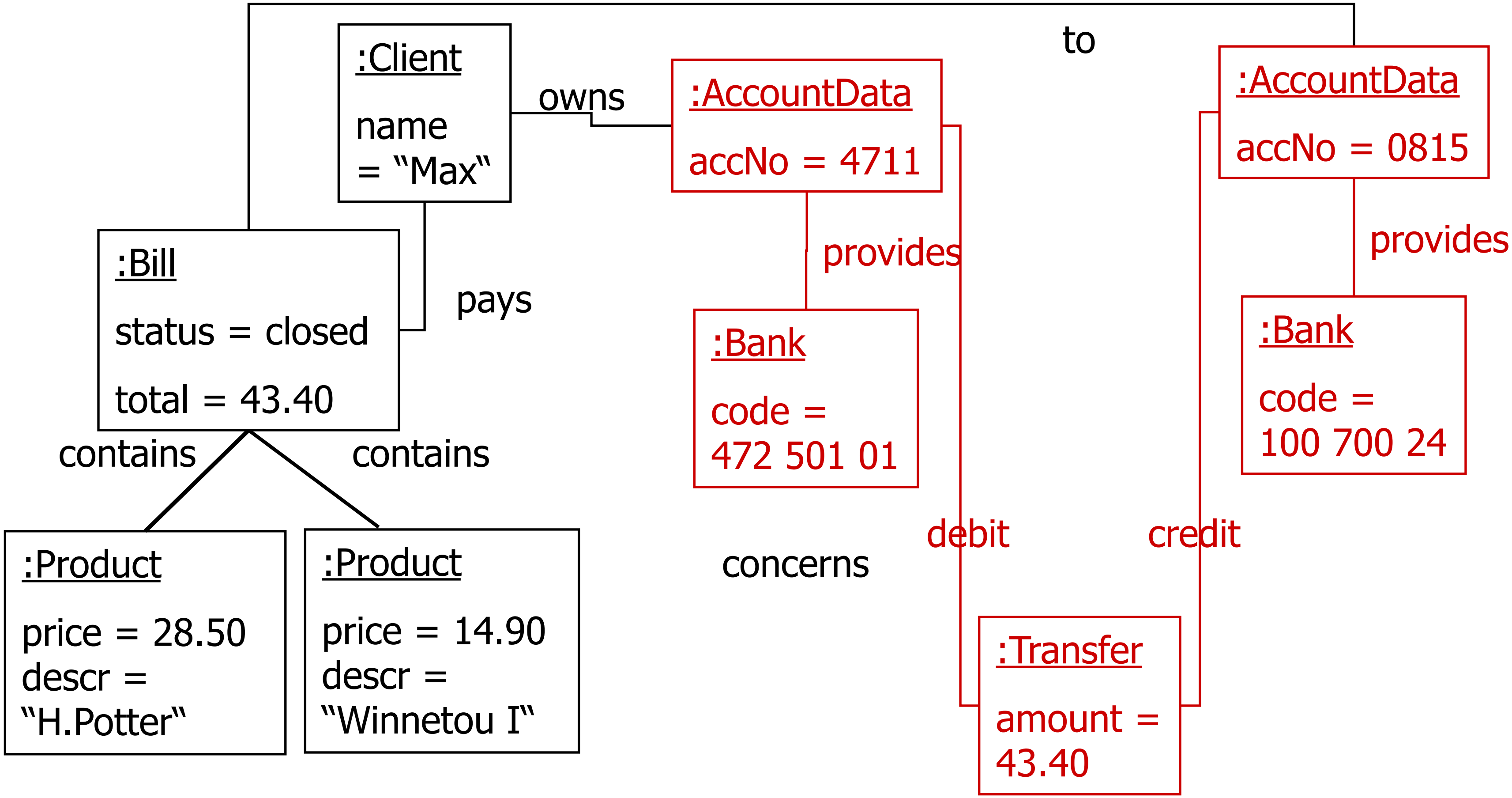
Instances:



*For all instances **a** over **A**, **b** over **B***

proj(trans(a)) = a *trans(proj(b))* $\subseteq$ *b*

Instances: Agent



:Client

name = “Max‟

owns

:AccountData

accNo = 4711

to

:AccountData

accNo = 0815

:Bill

status = closed

total = 43.40

pays

provides

:Bank

code = 472 501 01

provides

:Bank

code = 100 700 24

contains

contains

:Product

price = 28.50
descr = “H.Potter‟

:Product

price = 14.90
descr = “Winnetou I‟

concerns

debit

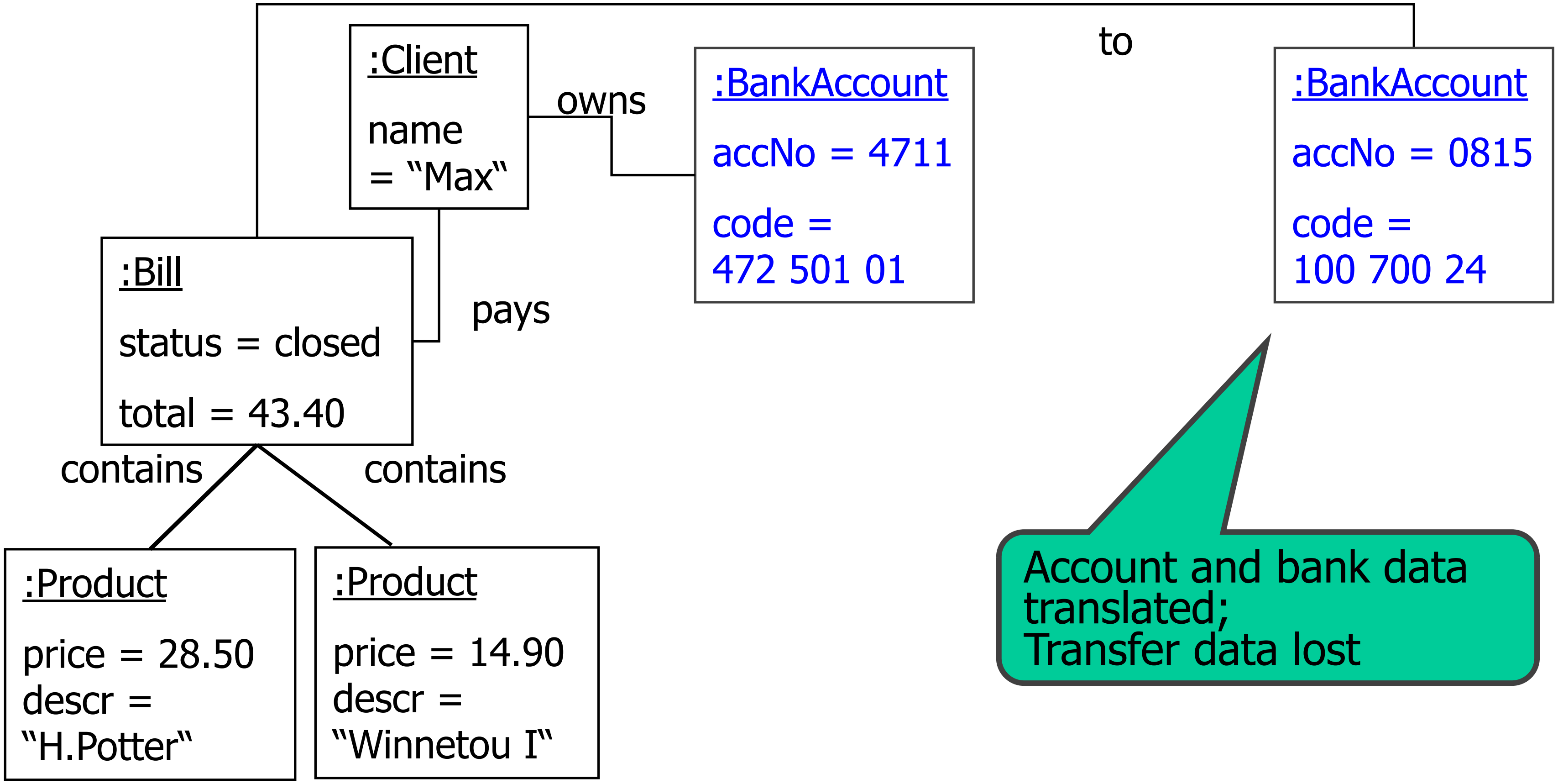
credit

:Transfer

amount = 43.40

Instances: Shop ← Agent

contravariant



:Client

name = “Max‟

owns

:BankAccount

accNo = 4711

code = 472 501 01

to

:BankAccount

accNo = 0815

code = 100 700 24

pays

:Bill

status = closed

total = 43.40

contains

contains

:Product

price = 28.50

descr = “H.Potter‟

:Product

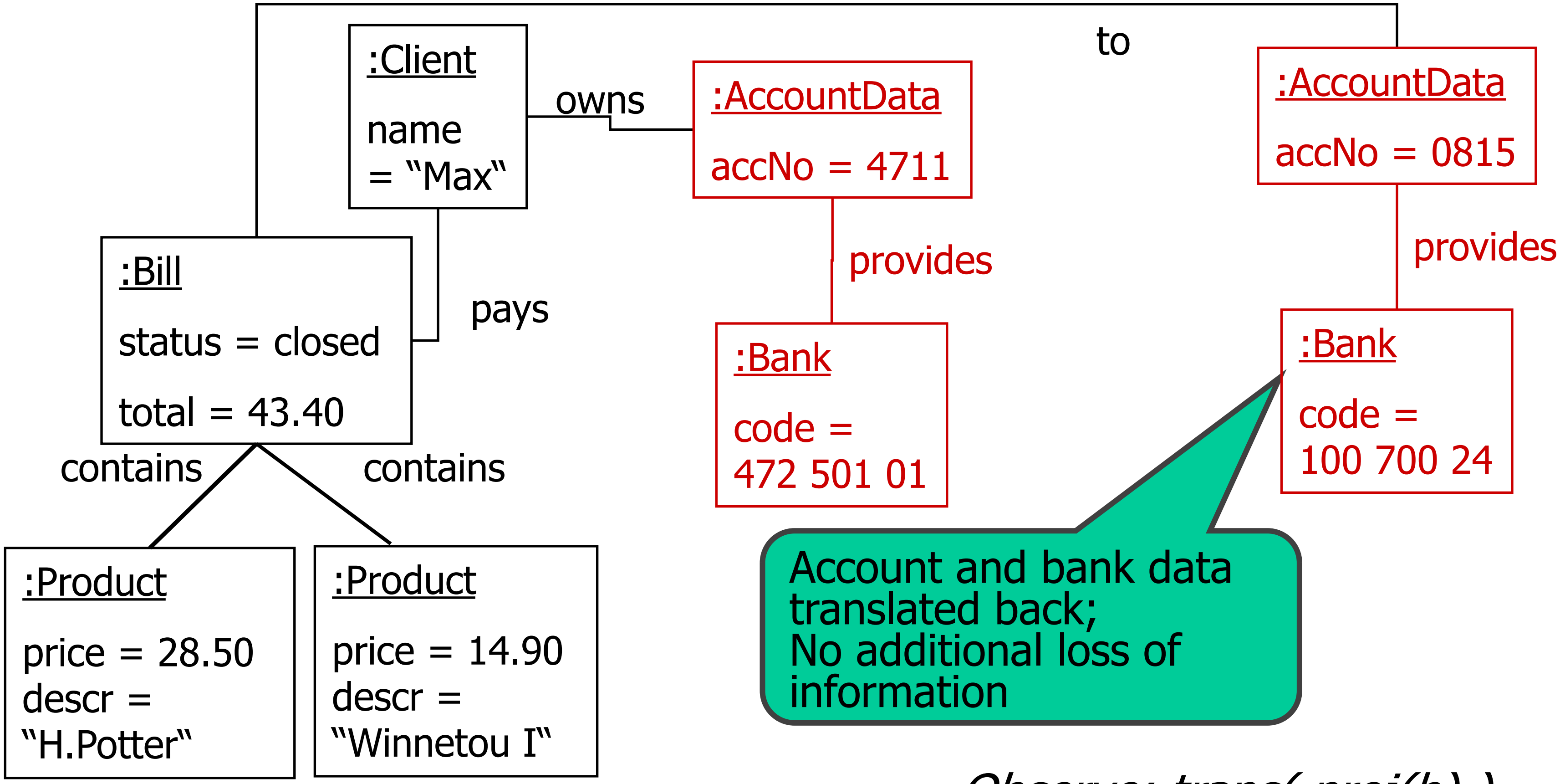
price = 14.90

descr = “Winnetou I‟

Account and bank data translated;
Transfer data lost

Instances: Shop → Agent

covariant



:Client

name = "Max"

owns

:AccountData

accNo = 4711

provides

:Bank

code = 472 501 01

to

:AccountData

accNo = 0815

provides

:Bank

code = 100 700 24

pays

:Bill

status = closed

total = 43.40

contains

contains

:Product

price = 28.50

descr = "H.Potter"

:Product

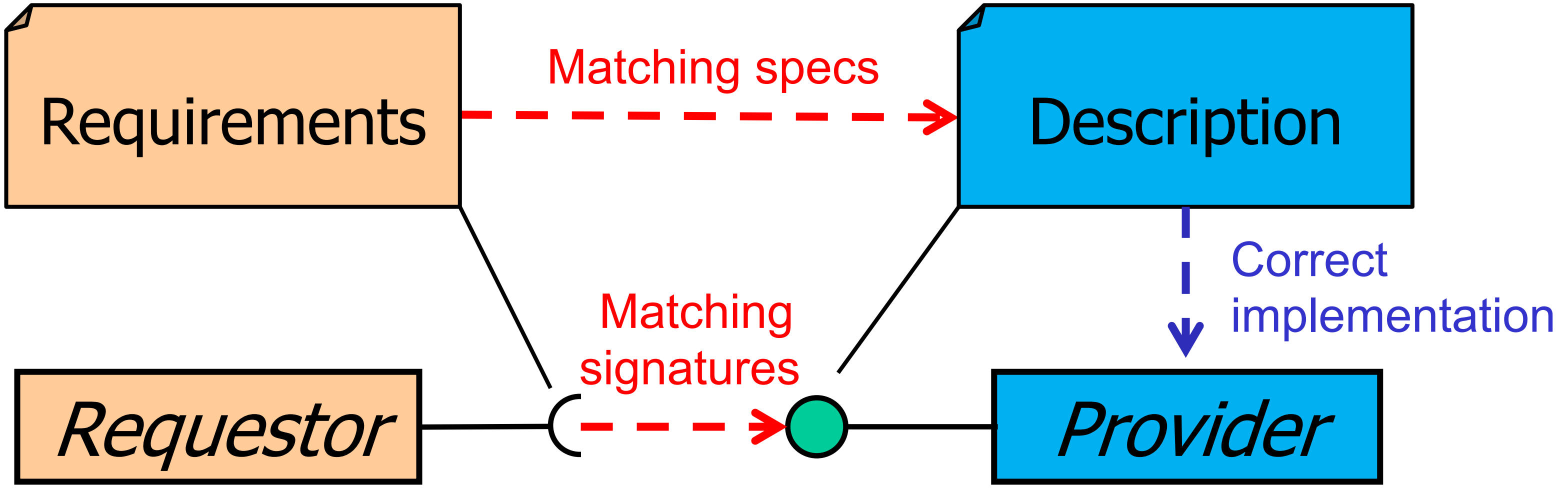
price = 14.90

descr = "Winnetou I"

Account and bank data translated back;
No additional loss of information

Observe: $trans(\ proj(b)\ ) \subseteq b$

Consistency in Service-oriented Systems



External: between required and provided specifications

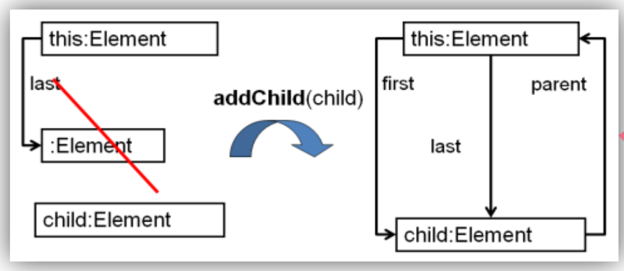
Matching data models and operations

Internal: between specification and implementation

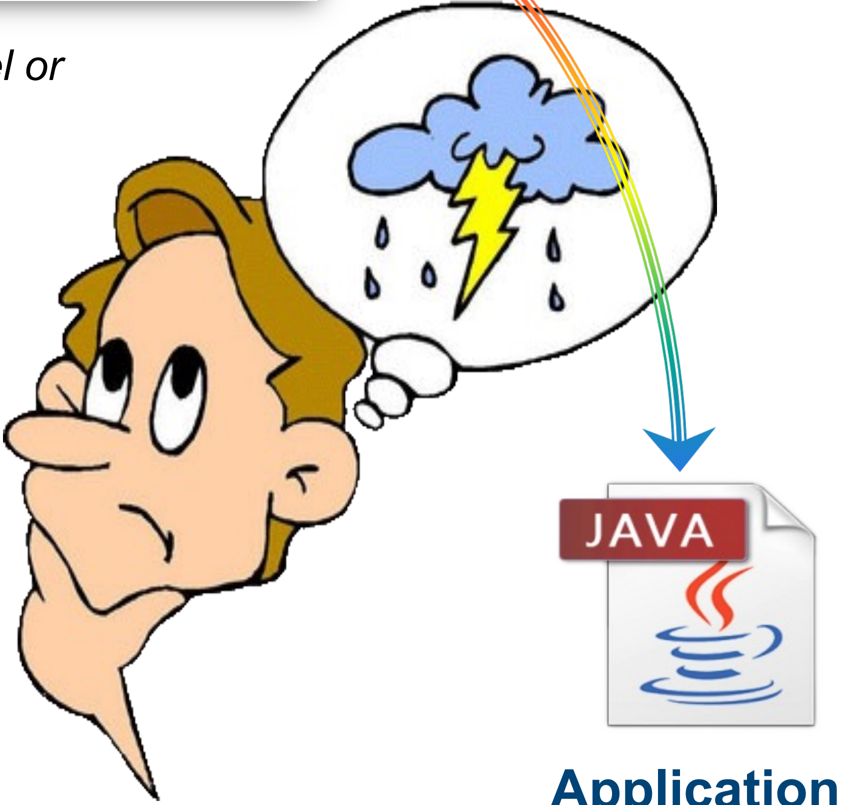
Testing and reverse engineering

Inferring Visual Contracts from Implementations

Visual Contract

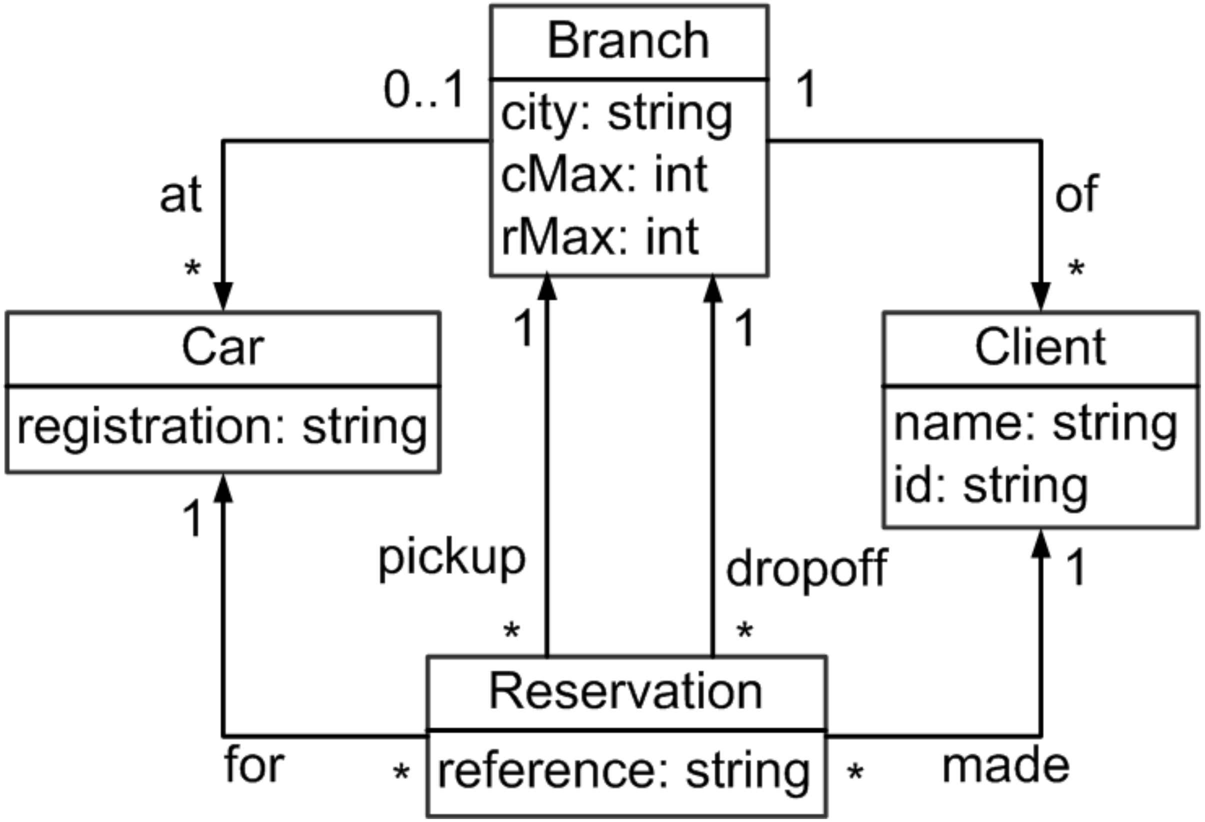


How to model or extract VCs?



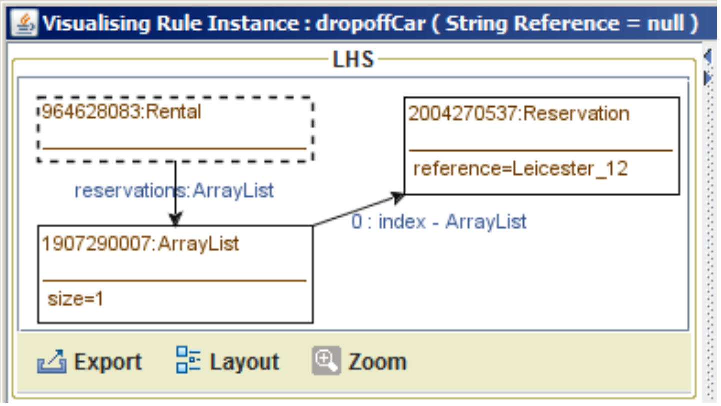
Application

Example: Car Rental Service

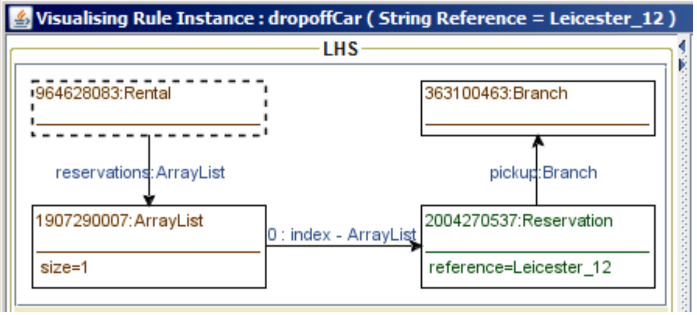


Code vs Visual Contracts

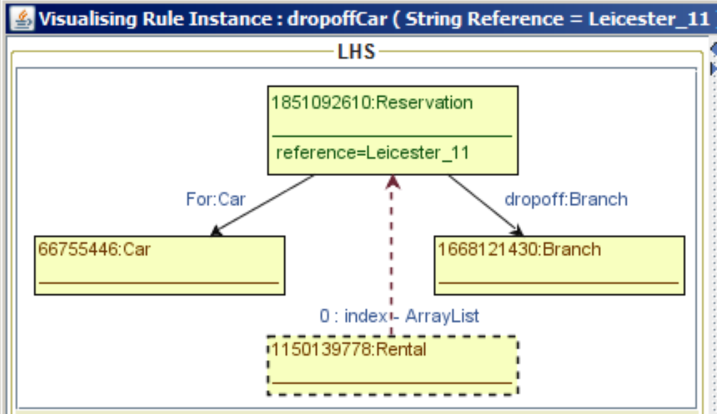
```
public void dropoffCar(String Reference){

    int iIndex = getReservationIndex(Reference);
    if (iIndex==−1){
        return;
    }

    Reservation getReservation = this.reservations.get(iIndex);
    // check if reserved car has been picked up already
    if (getReservation.pickup!=null){
        return;
    }

    // return reserved car to the drop−off branch
    getReservation.dropoff.at.add(getReservation.for);
    // remove reservation object
    this.reservations.remove(iIndex);
}
```



Visualising Rule Instance : dropoffCar (String Reference = null)

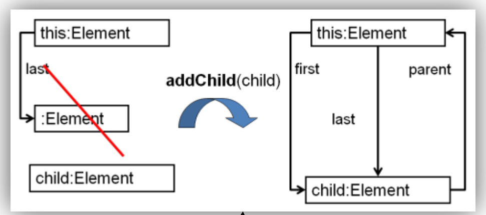


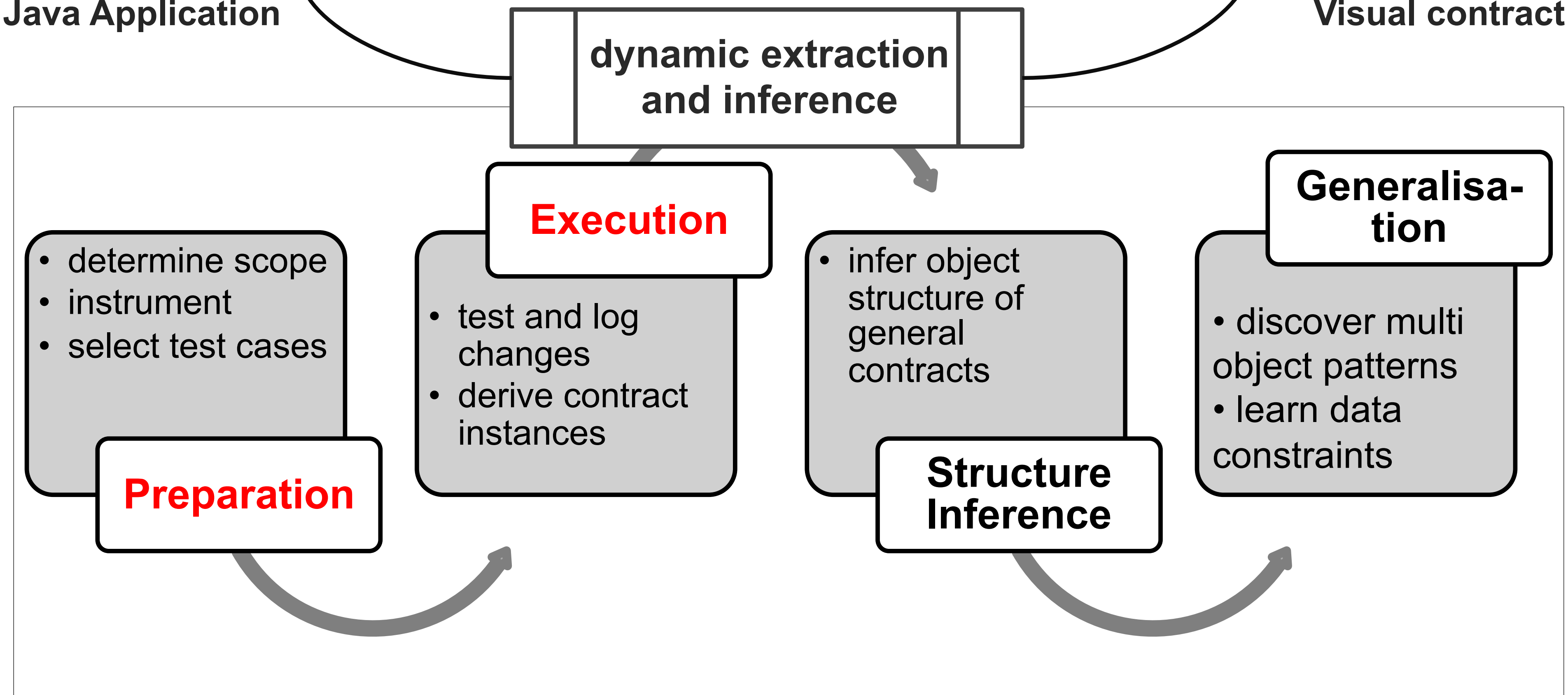
Visualising Rule Instance : dropoffCar (String Reference = Leicester_12)



Visualising Rule Instance : dropoffCar (String Reference = Leicester_11

Reverse Engineering Visual Contracts





Java Application

Visual contract

dynamic extraction and inference

Preparation

- determine scope
- instrument
- select test cases

Execution

- test and log changes
- derive contract instances

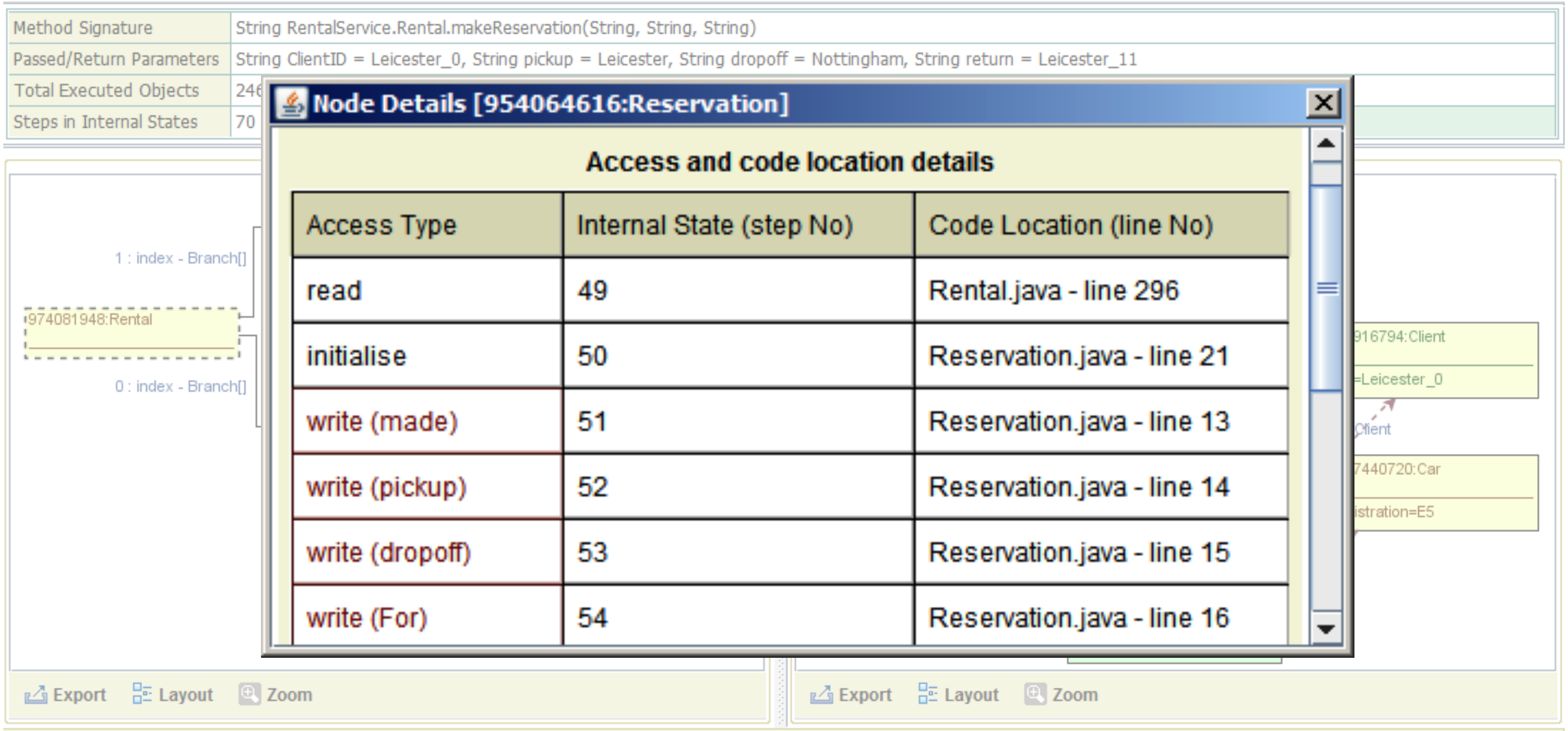
Structure Inference

- infer object structure of general contracts

Generalisa-tion

- discover multi object patterns
- learn data constraints

Test and Log Changes



Method Signature	String RentalService.Rental.makeReservation(String, String, String)
Passed/Return Parameters	String ClientID = Leicester_0, String pickup = Leicester, String dropoff = Nottingham, String return = Leicester_11
Total Executed Objects	246
Steps in Internal States	70

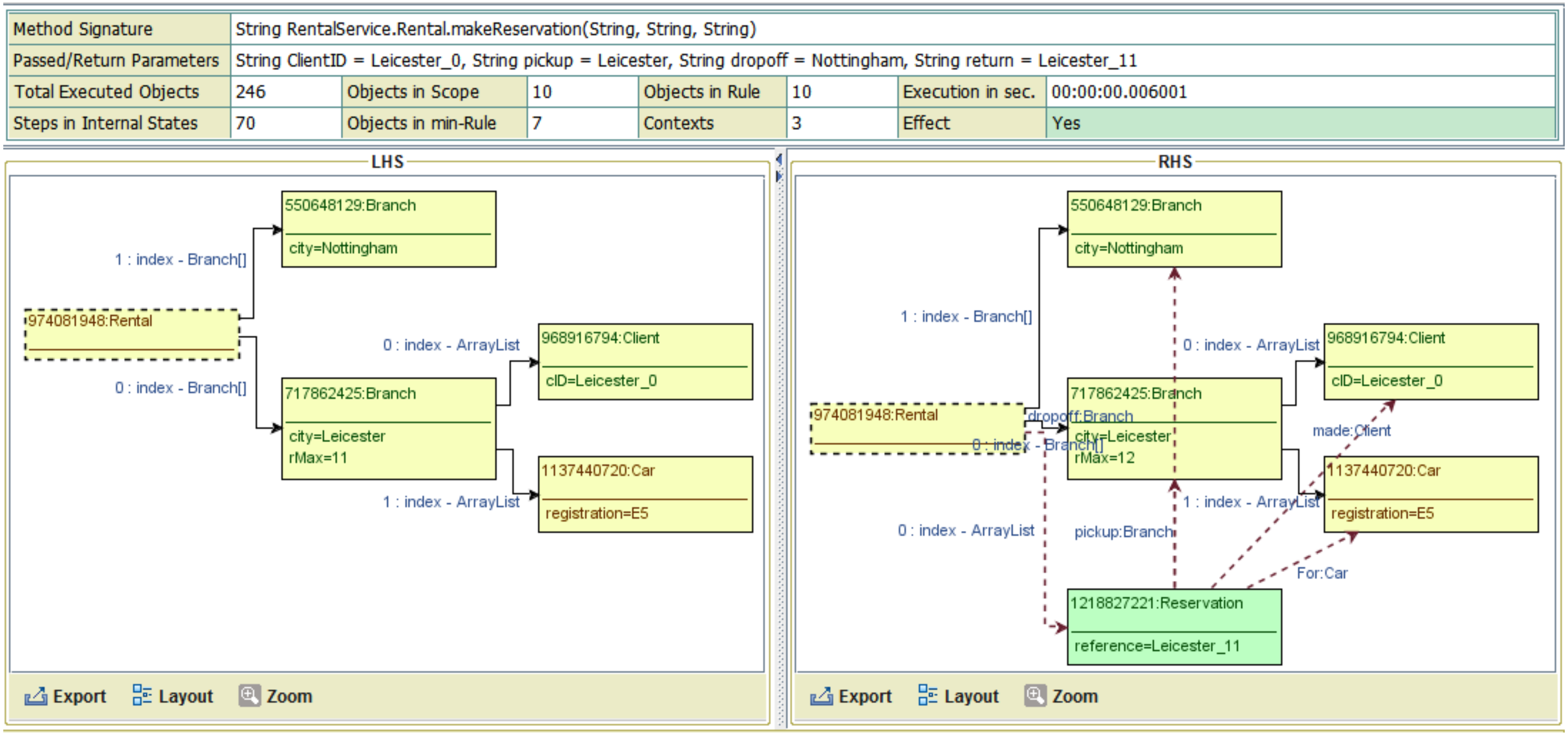
Node Details [954064616:Reservation]

Access and code location details

Access Type	Internal State (step No)	Code Location (line No)
read	49	Rental.java - line 296
initialise	50	Reservation.java - line 21
write (made)	51	Reservation.java - line 13
write (pickup)	52	Reservation.java - line 14
write (dropoff)	53	Reservation.java - line 15
write (For)	54	Reservation.java - line 16

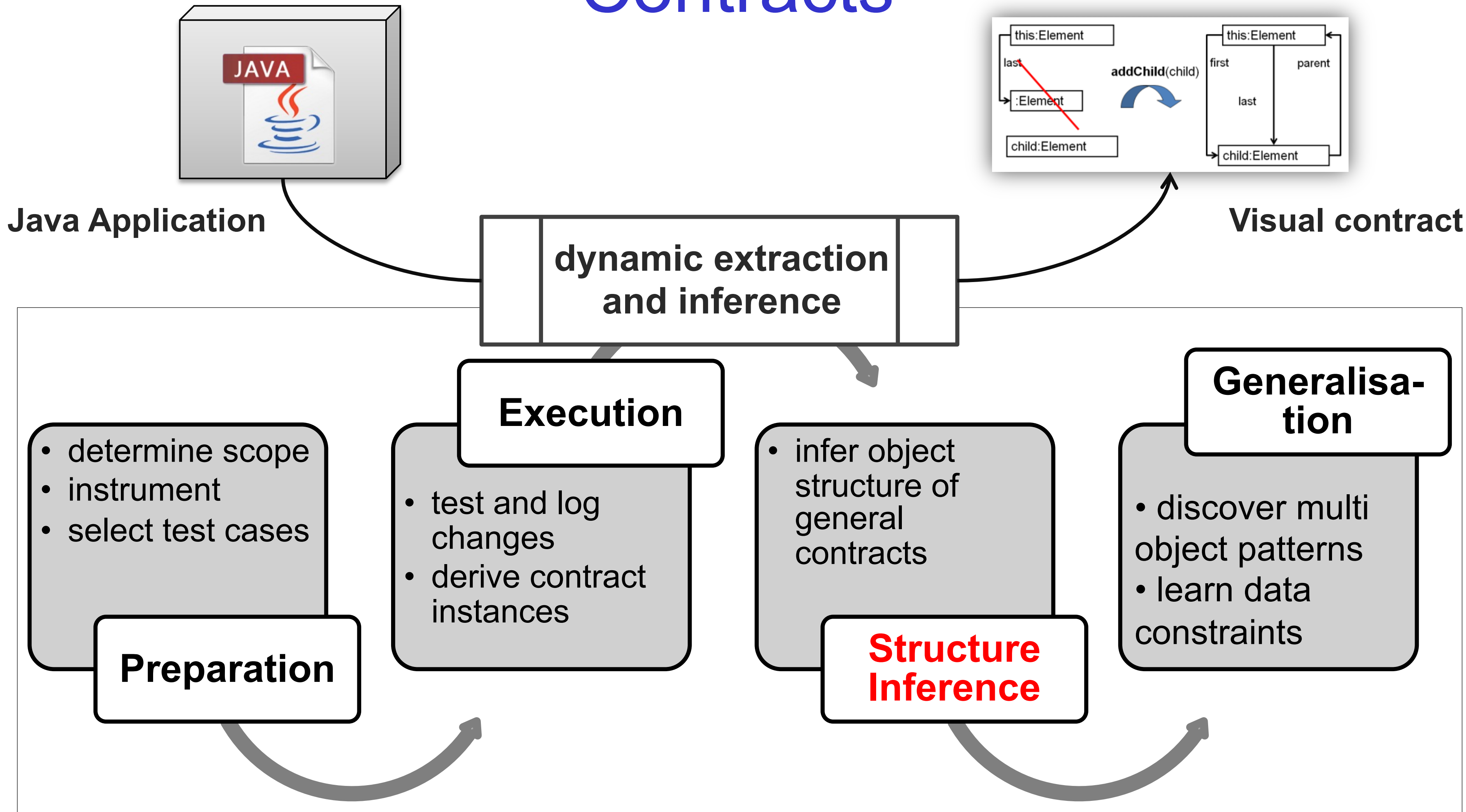
By clicking on a node element

Deriving Contract Instances



Method Signature	String RentalService.Rental.makeReservation(String, String, String)						
Passed/Return Parameters	String ClientID = Leicester_0, String pickup = Leicester, String dropoff = Nottingham, String return = Leicester_11						
Total Executed Objects	246	Objects in Scope	10	Objects in Rule	10	Execution in sec.	00:00:00.006001
Steps in Internal States	70	Objects in min-Rule	7	Contexts	3	Effect	Yes

Reverse Engineering Visual Contracts



Java Application

Visual contract

dynamic extraction and inference

Preparation

- determine scope
- instrument
- select test cases

Execution

- test and log changes
- derive contract instances

Structure Inference

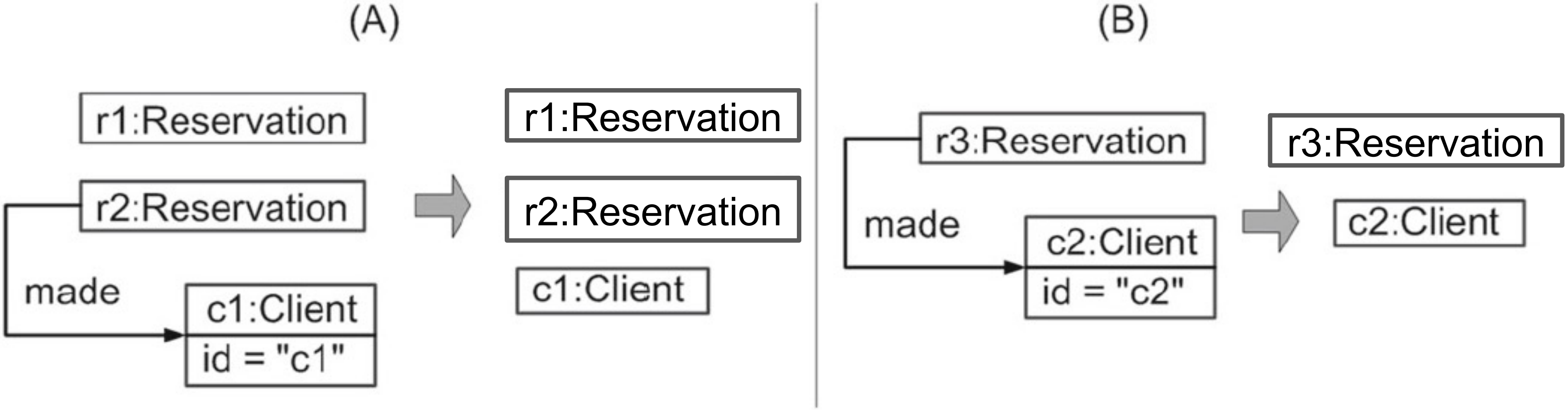
- infer object structure of general contracts

Generalisation

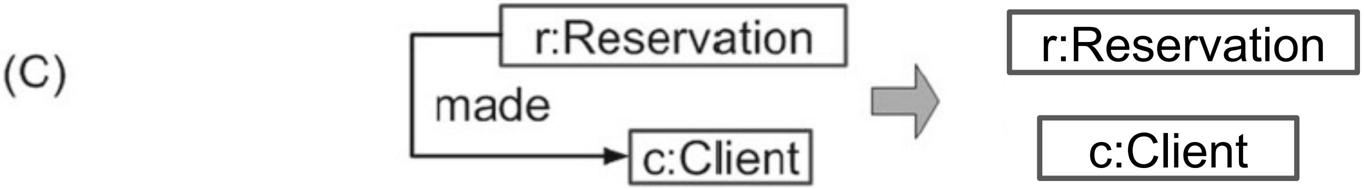
- discover multi object patterns
- learn data constraints

Minimal Contracts and Shared Context

Two contract-instances extracted from : *cancelClientReservation(..)*



The maximal rule extracted from : *(A)* and *(B)*



Minimal Contracts and Shared Context

From all instances representing executions of the same operation generate

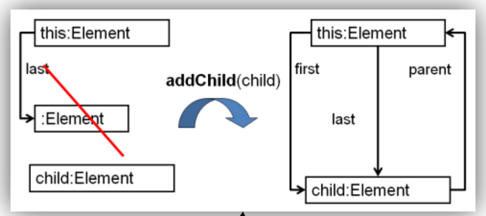
Minimal rule

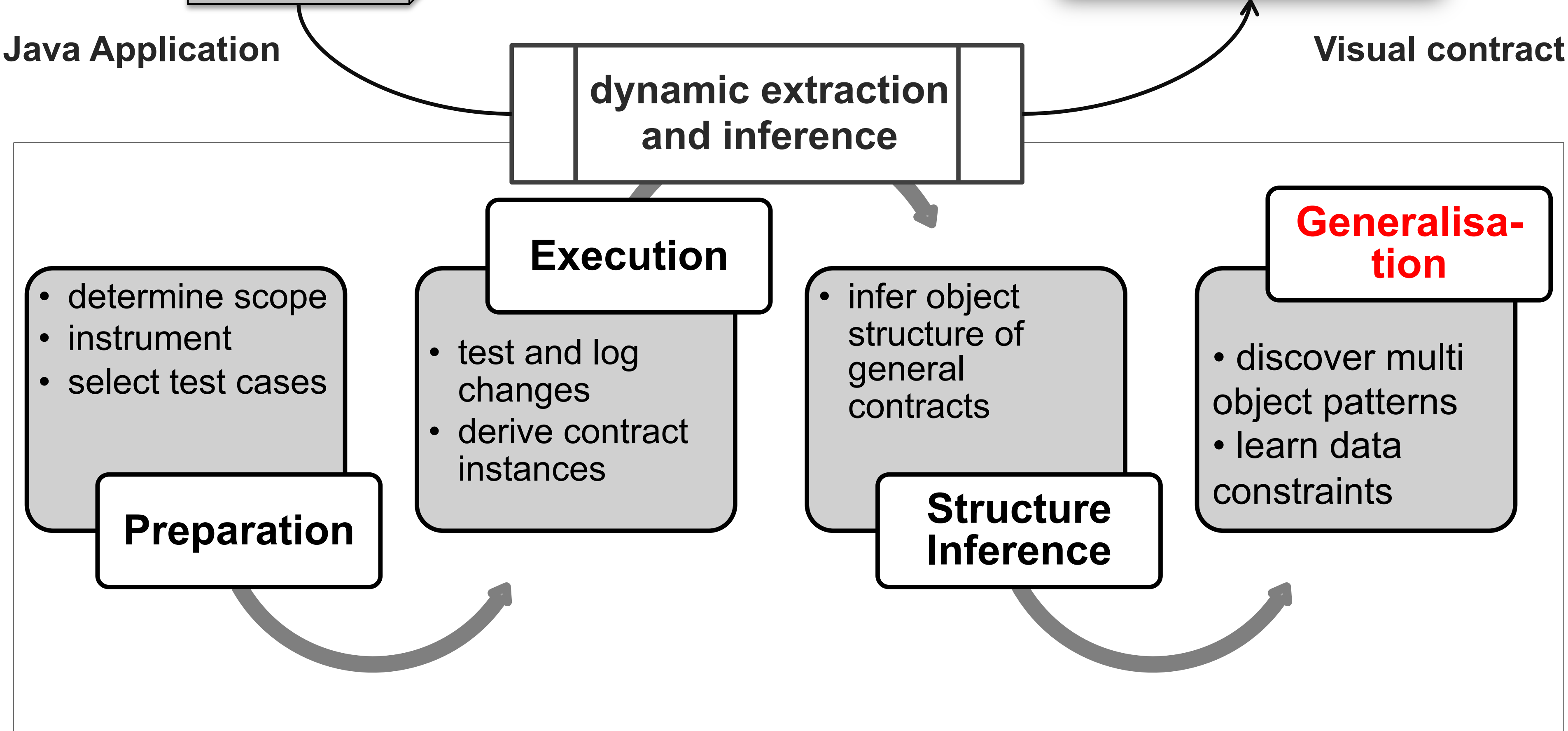
- smallest rule able to perform the observed object transformation
- cut all context not needed to achieve observed changes nor required as input or return
- use to classify instances by effect: all instances with the same minimal rule have the same effect, but possibly different preconditions

Maximal rule

- extend the minimal rule by all context present in all instances

Reverse Engineering Visual Contracts





Java Application

Visual contract

dynamic extraction and inference

Preparation

- determine scope
- instrument
- select test cases

Execution

- test and log changes
- derive contract instances

Structure Inference

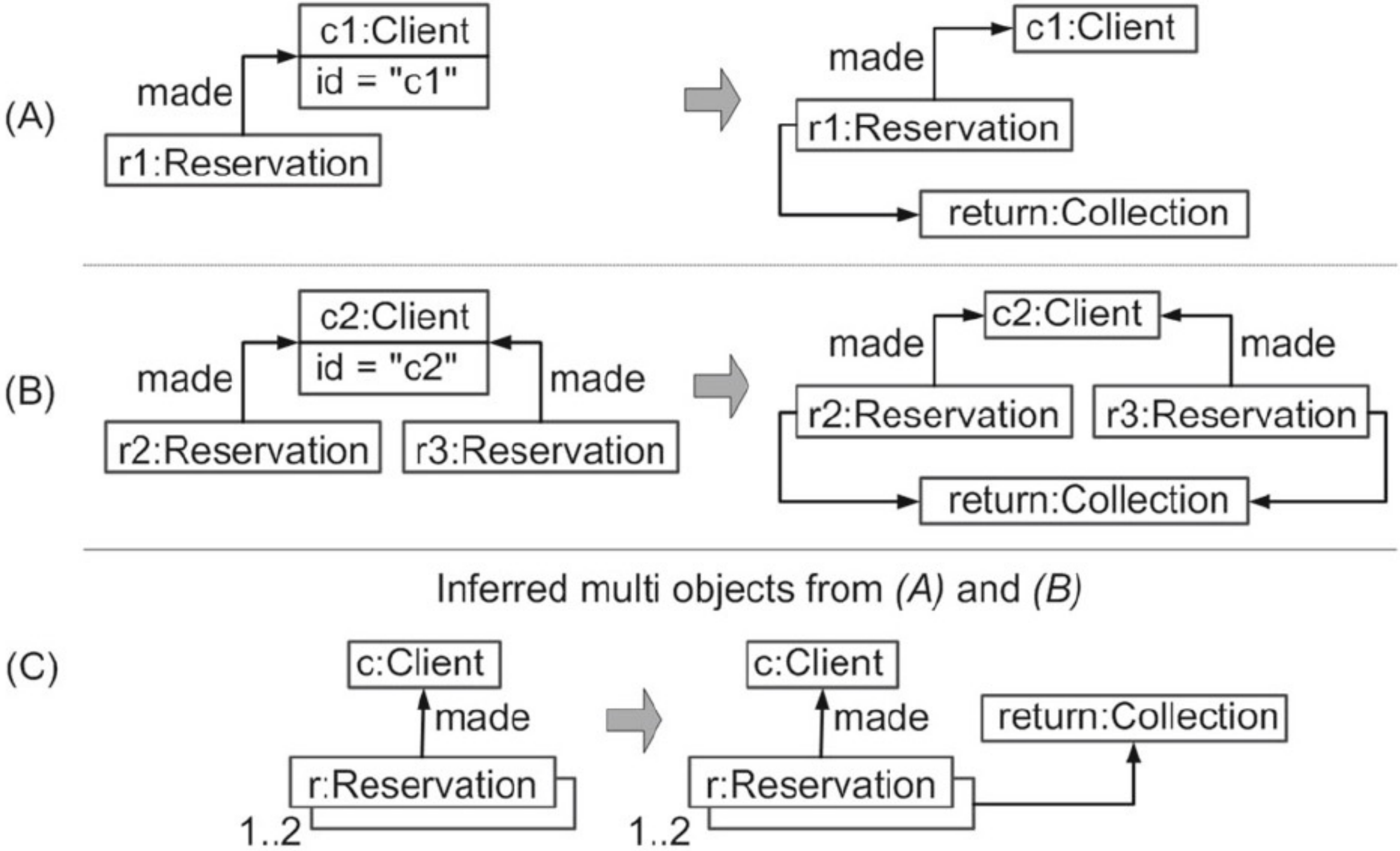
- infer object structure of general contracts

Generalisa-tion

- discover multi object patterns
- learn data constraints

Multi Object Patterns

Two instances extracted from : *showClientReservation(..)= returnList*

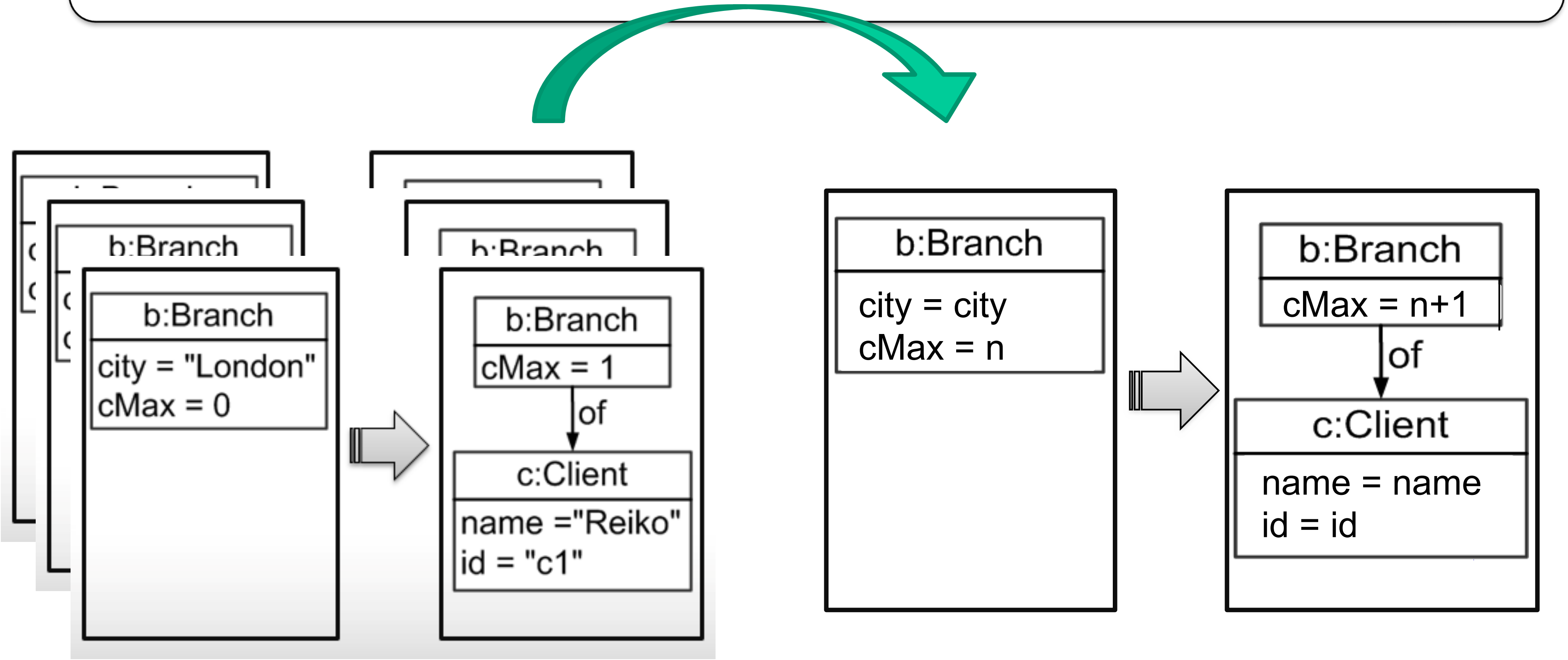


Inferred multi objects from *(A)* and *(B)*

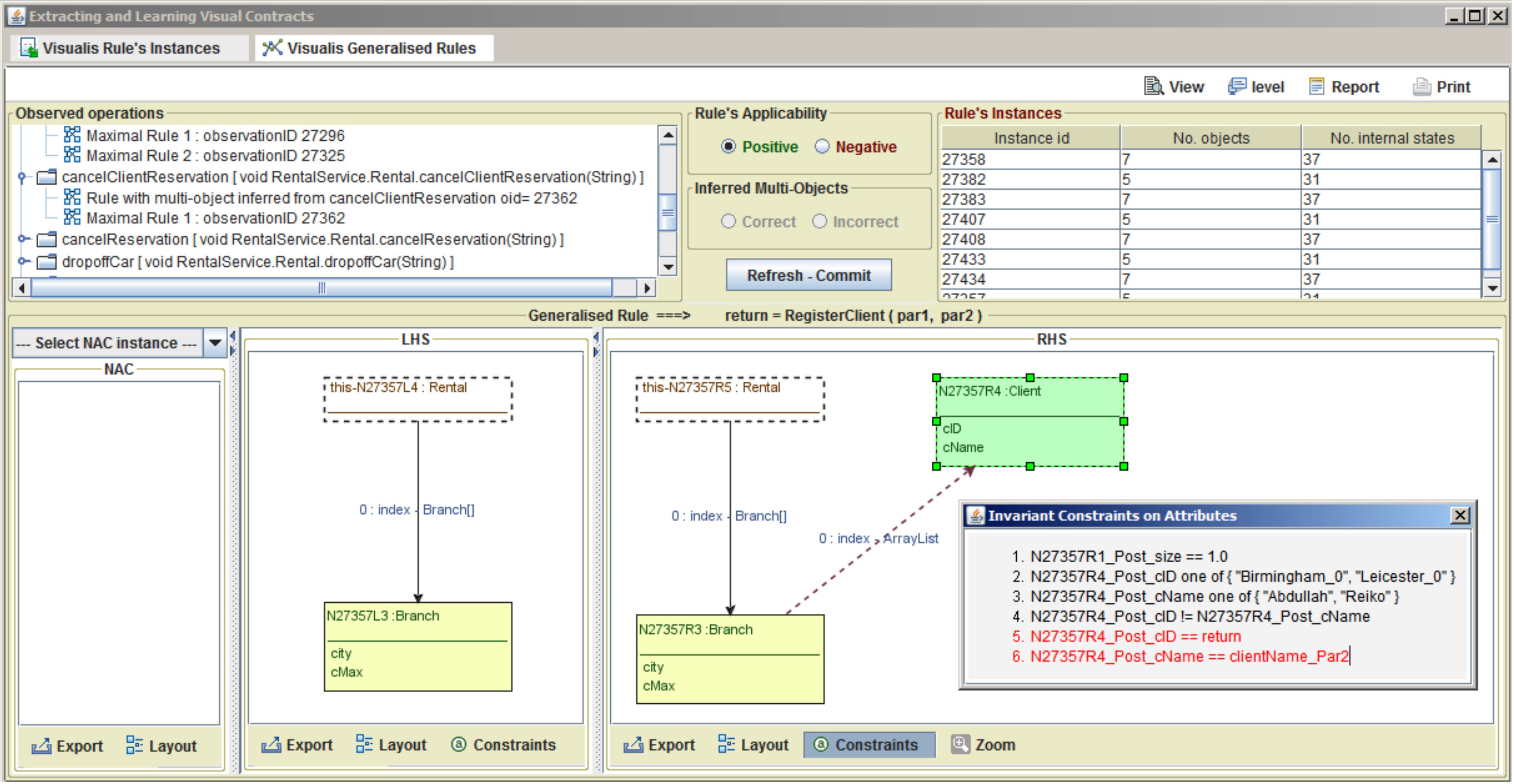
Data Constraints

Consider actual data values extracted from rule instances.

Discover invariant conditions over attributes and (data) parameters.



Generalised Contract



- Maximal Rule 1 : observationID 27296
- Maximal Rule 2 : observationID 27325
- cancelClientReservation [void RentalService.Rental.cancelClientReservation(String)]
  - Rule with multi-object inferred from cancelClientReservation oid= 27362
  - Maximal Rule 1 : observationID 27362
- cancelReservation [void RentalService.Rental.cancelReservation(String)]
- dropoffCar [void RentalService.Rental.dropoffCar(String)]

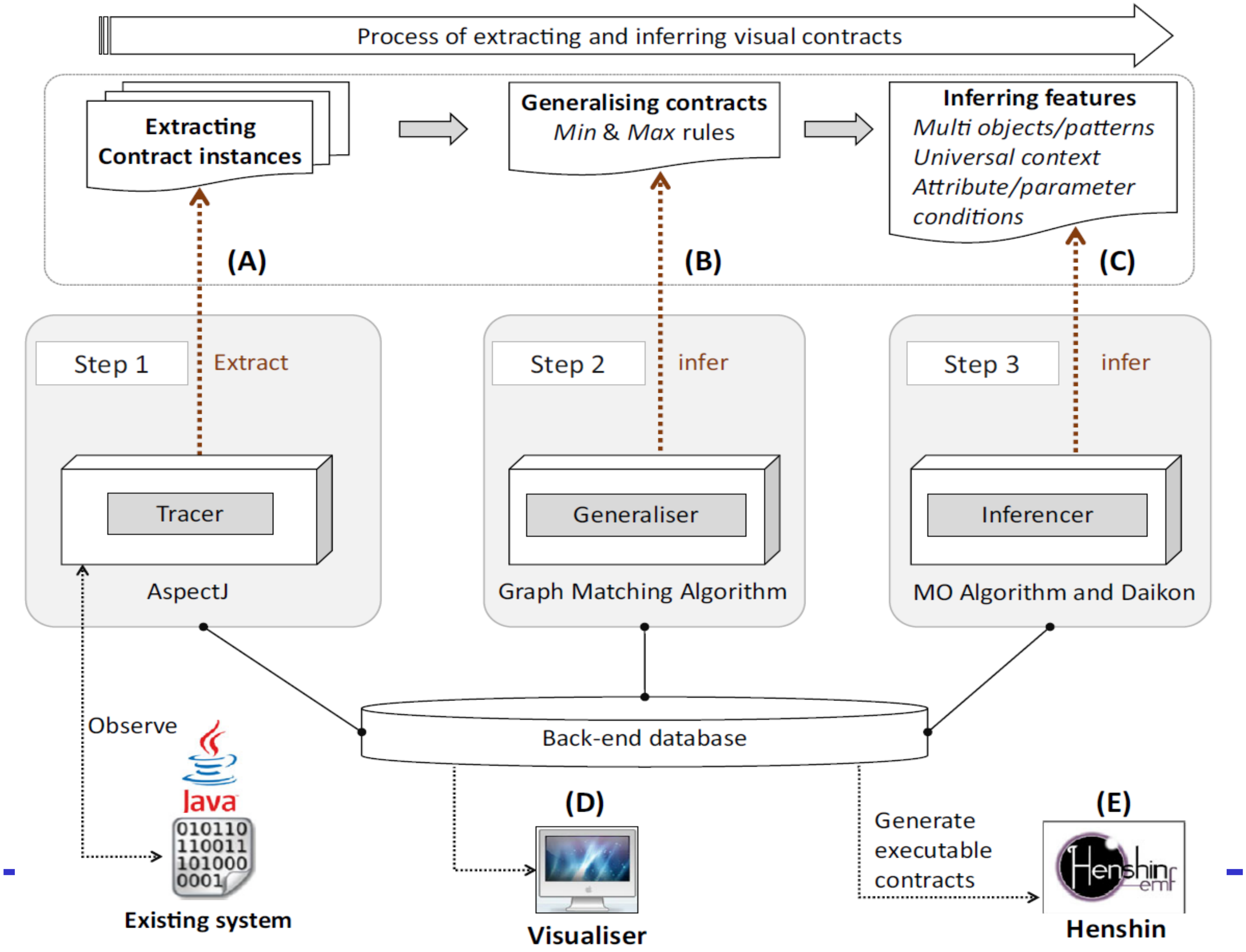
Instance id	No. objects	No. internal states
27358	7	37
27382	5	31
27383	7	37
27407	5	31
27408	7	37
27433	5	31
27434	7	37

Generalised Rule ===> return = RegisterClient (par1, par2)

Invariant Constraints on Attributes

1. N27357R1_Post_size == 1.0
2. N27357R4_Post_cID one of { "Birmingham_0", "Leicester_0" }
3. N27357R4_Post_cName one of { "Abdullah", "Reiko" }
4. N27357R4_Post_cID != N27357R4_Post_cName
5. N27357R4_Post_cID == return
6. N27357R4_Post_cName == clientName_Par2

The Visual Contract Extractor (VCE) Tool



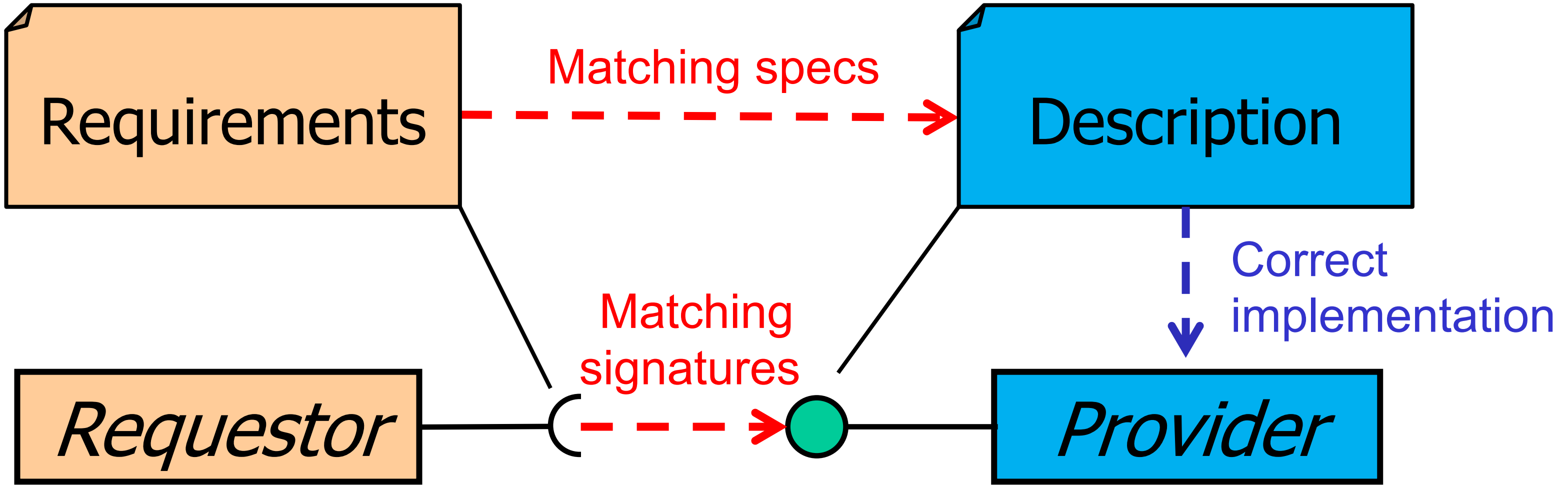
Experimental Evaluation

- Completeness and correctness of extracted contracts
  - *Based on dynamic analysis* ➔ *no completeness*

    Higher code coverage leads to more complete model.
  - *Partial logging scope* ➔ *over-approximation*

    Larger scope leads to more stronger preconditions and effects.
- Utility of visual contracts
  - *User study with **66** participating students*
- Scalability of contract extraction
  - *NanoXML and JHotDraw case studies*

Consistency in Service-oriented Systems



External: between required and provided specifications

Matching data models and operations

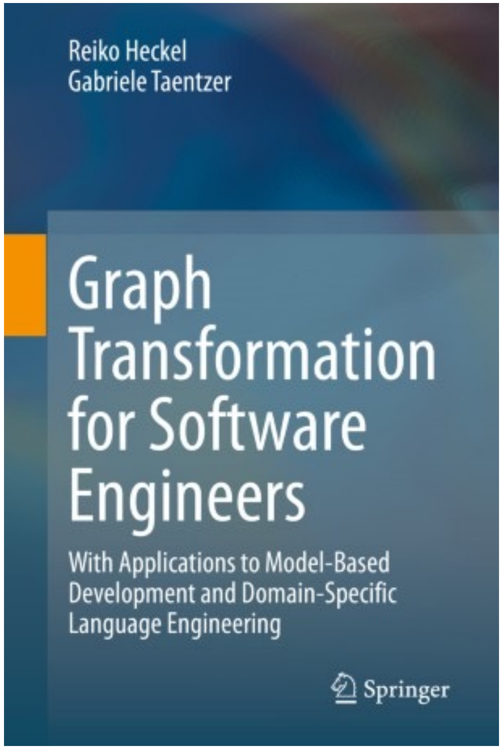
Internal: between specification and implementation

Testing and reverse engineering

Session 1

Part 1: Introduction to Graph Transformation

1. Graphs for Modelling and Specification
2. Graph Transformation Concepts
3. Beyond Individual Rules: Usage Scenarios and Control Structures
4. Analysis and Improvement of Graph Transformation Systems


Reiko Heckel
Gabriele Taentzer

Graph Transformation for Software Engineers

With Applications to Model-Based Development and Domain-Specific Language Engineering

Springer

The book is available from Springer
https://link.springer.com/book/10.1007/978-3-030-43916-3
A free authors' copy and further material is available here:
http://graph-transformation-for-software-engineers.org/

Part 2: Graph Transformation in Software Engineering

Session 2

5. Detecting Inconsistent Requirements in a Use-Case-Driven Approach

Session 3

6. Service Specification and Matching
7. Model-Based Testing
8. Reverse Engineering: Inferring Visual Contracts from Java Programs

9. Stochastic Analysis of Dynamic Software Architectures
10. Advanced Modelling-Language Definition: Integrating Metamodelling with Graph Transformation
11. Improving Models and Understanding Model Changes
12. Translating and Synchronising Models