 University of Leicester


Model-based Development of Web Services using Design-by-Contract

Reiko Heckel
University of Leicester, UK

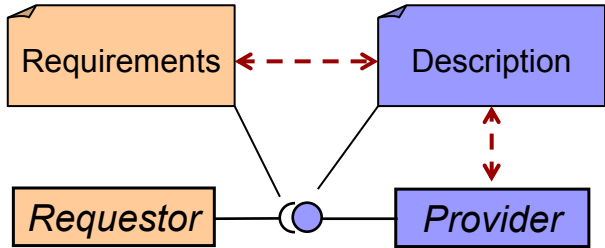
Joint work with M. Lohmann, A. Cherchago,
J.H. Hausmann, Paderborn

TFS Colloquium, TU Berlin, 5. 12. 2005

1

 University of Leicester

Consistency of Service Composition



1. **External:** between interface specifications
2. **Internal:** between interface specification and implementation

University of Leicester

Example: Car Rental Service

<<interface>>
RentalServiceRequired

reservCar(c:Customer, car:Car, ri:RentalInfo)
...

<<interface>>
RentalServiceProvided

makeReserv(c:Customer, car:Car, ri:RentalInfo): EContract
...

Matching *provider* and *requestor* specification within *registry* must ensure compatibility of

- Data types
 - Does Customer have the same meaning for requestor and provider?
- Operation signatures
 - Can provider operation be supplied with suitable parameters from a call of requestor operation?
- Behavior
 - Does provided operation actually carry out what is expected by a requestor?

University of Leicester

Data Types and Signatures

<<interface>>
RentalServiceRequired

reservCar(c:Customer, car:Car, ri:RentalInfo)
...

<<interface>>
RentalServiceProvided

makeReserv(c:Customer, car:Car, ri:RentalInfo): EContract
...

- Reorder and rename pars
- Skip input of requestor
- Ignore output of provider

Data types: parties use common domain model (ontology)

Operation signatures: Zaremski and Wing: *Signature matching: A tool for using software libraries.* TOSEM 1995.

```


classDiagram
    class Customer
    class RentalInfo {
        pic-upDate: Date
        returnDate: Date
        location: String
    }
    class EContract {
        isSigned: Bool
    }
    class Vehicle {
        Id: String
    }
    class Car
    class Truck
    class Van

    Customer --> RentalInfo : provides
    Customer --> EContract : signs
    RentalInfo --> EContract : for
    EContract --> Vehicle : for
    Vehicle <|-- Car
    Vehicle <|-- Truck
    Vehicle <|-- Van
    Customer --> Vehicle : reserves
    
```

University of Leicester

Behavior: Operation Contracts

Pre-condition:
Customer provides rental info and selects car



Effect:
Car is reserved for customer

Required

- Formal specification (logic, graph transformation, ...) for automatic matching
- Integration into mainstream SW development methods (UML) for wider applicability

Outline

- Contracts as graph transformation rules
- Semantics of rules
- Semantic / syntactic compatibility, soundness

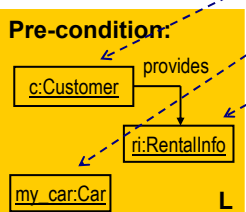
University of Leicester

Contracts as Graph Transformation Rules


Signature: `reservCar(c:Customer, my_car:Car, ri:RentalInfo)`

Behavior: GT rule

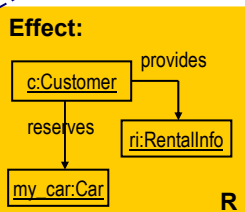
Pre-condition:



L



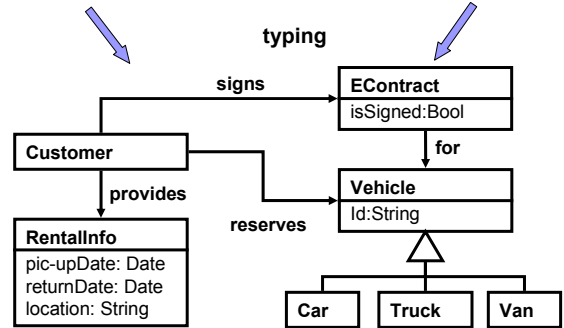
Effect:



R

Typed DPO [Corradini et al 96]

Data types: type graph



University of Leicester

What is the right notion of compatibility? That depends on ...

how services should interact:

1. Requestor guarantees pre_R
→ Provider assumes pre_P
2. Provider guarantees $effect_P$
→ Requestor assumes $effect_R$

... a *contravariant* relation.

what it should mean, that:

- an *assumption* is correct
- a *guarantee* is fulfilled

... a question about the *semantics of contracts*.

University of Leicester

Operational Semantics: The DPO Approach

- L is embedded into graph G .
- The elements of G matched by $L - l(K)$ are removed.
- The elements matched by $R - r(K)$ are added to D .

The changes to G are **exactly** those specified by the rule

University of Leicester

Loose Semantics of Contracts

Requestor has only loose idea of behavior of the other service

→ Contracts are incomplete specifications of service behavior

1. call
2. return

Provider has complete info, but may prefer not to publish everything

$$\begin{array}{ccccc}
 & L & \xleftarrow{l} & K & \xrightarrow{r} & R \\
 d_L \downarrow & & (PB) & d_K \downarrow & (PB) & d_R \downarrow \\
 & G & \xleftarrow{g} & D & \xrightarrow{h} & H
 \end{array}$$

Formally: Double-Pullback (DPB), allows unspecified

Deletion: *at least* elements of G matched by $L - l(K)$ are removed

Creation: *at least* elements matched by $R - r(K)$ are added to D

(faithful) transition vs. transformation

University of Leicester

Contracts as Rules, revisited

→ Positive Application Conditions

Precondition: what must be

- present before, no matter what happens later

Effect: what must be

- deleted
- preserved
- created

States: L, K, R, G, D, H

Transitions: $L \xrightarrow{l} K$, $K \xrightarrow{r} R$, $L \xrightarrow{d_L} G$, $K \xrightarrow{d_K} D$, $R \xrightarrow{d_R} H$

Labels: (PB) for $L \xrightarrow{l} K$ and $K \xrightarrow{r} R$

University of Leicester

What is the right notion of compatibility? That depends on ...

how services should interact:

1. Requestor **guarantees** pre_R
→ Provider **assumes** pre_P
2. Provider **guarantees** $effect_P$
→ Requestor **assumes** $effect_R$

... a *contravariant* relation.

what it should mean, that:

- an *assumption* is correct
- a *guarantee* is fulfilled

... a question about the *semantics of contracts*. ✓

University of Leicester

Semantic Compatibility

R:

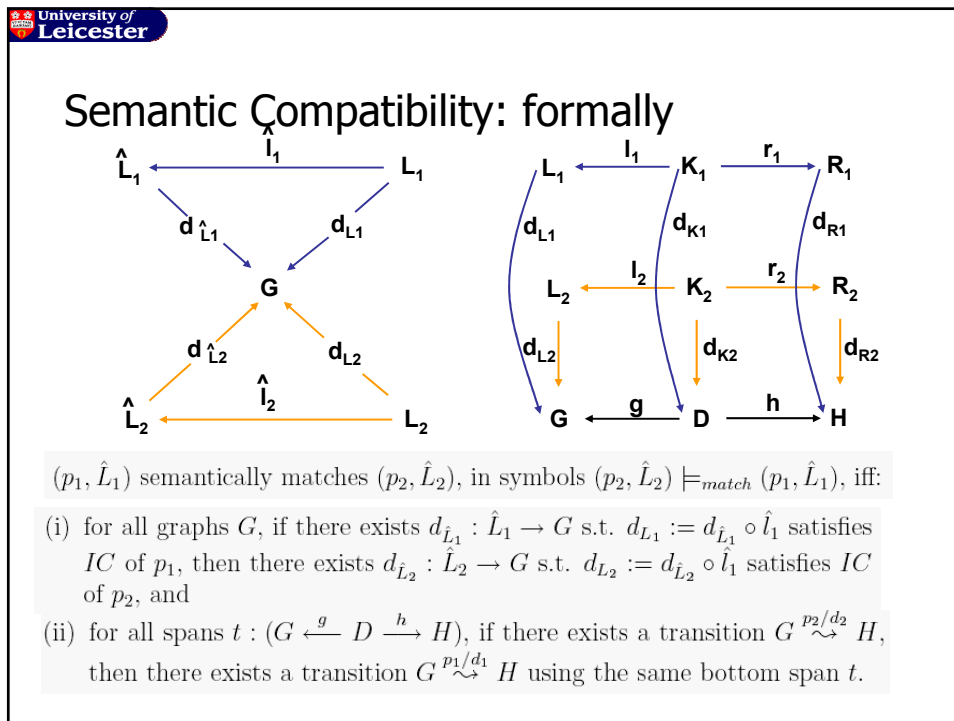
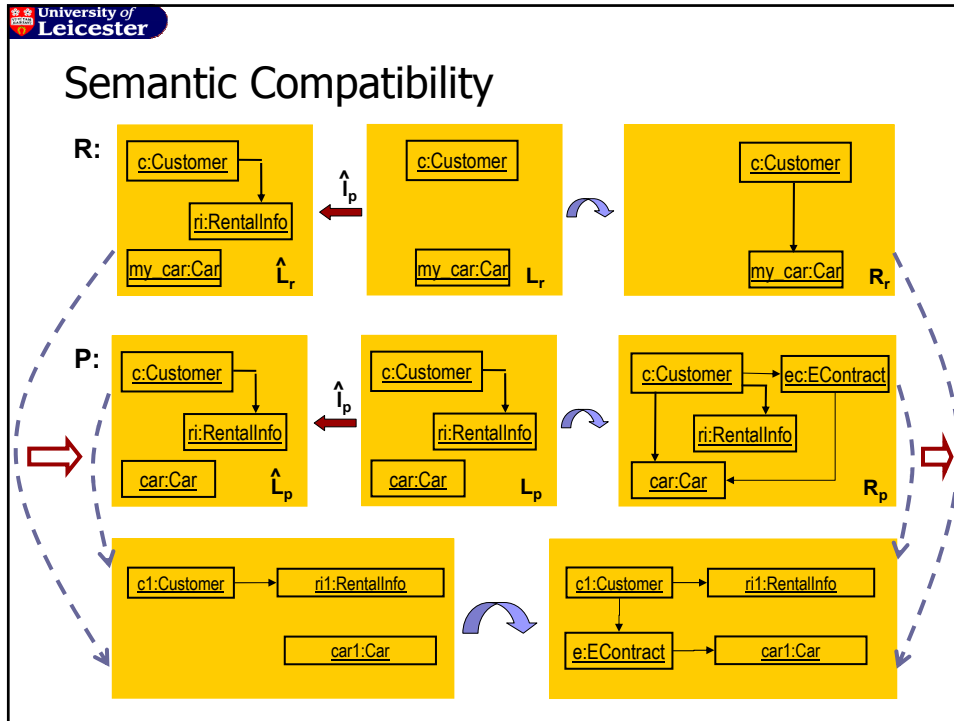
- Initial: $c:Customer$ → $ri:RentalInfo$ → $my_car:Car$ (with \hat{L}_r below $my_car:Car$)
- Intermediate: $c:Customer$ → $my_car:Car$ (with L_r below $my_car:Car$)
- Final: $c:Customer$ → $my_car:Car$ (with R_r below $my_car:Car$)

P:

- Initial: $c:Customer$ → $ri:RentalInfo$ → $car:Car$ (with \hat{L}_p below $car:Car$)
- Intermediate: $c:Customer$ → $ri:RentalInfo$ → $car:Car$ (with L_p below $car:Car$)
- Final: $c:Customer$ → $ec:EContract$ → $ri:RentalInfo$ → $car:Car$ (with R_p below $car:Car$)

Final R state: $c1:Customer$ → $ri1:RentalInfo$ → $car1:Car$

1. $pre_R \rightarrow pre_P$: applicability of requestor rule **implies** applicability of provider rule
2. $effect_P \rightarrow effect_R$: transition via provider rule **is also** transition via requestor rule.



University of Leicester

What do we have?

Semantic compatibility relation $| =$

- * quantified over all graphs and transitions
- * cannot be verified directly

Objective: syntactic matching relation $| \dashv \dashv$

- Soundness: $p_2 | \dashv \dashv p_1$ implies $p_2 | = p_1$
- Completeness: $p_2 | = p_1$ implies $p_2 | \dashv \dashv p_1$

University of Leicester

Syntactic Matching Relation

The diagram shows two rows of graphs, R (Requestor) and P (Provider), each with three nodes: `c:Customer`, `ri:RentalInfo`, and `my_car:Car` (or `car:Car`). The nodes are connected by arrows representing relationships. In R, `c:Customer` points to `ri:RentalInfo`, and `ri:RentalInfo` points to `my_car:Car`. In P, `c:Customer` points to `ri:RentalInfo`, and `ri:RentalInfo` points to `car:Car`. Additionally, in P, `c:Customer` points to `ec:EContract`, which in turn points to `ri:RentalInfo` and `car:Car`.

Red arrows labeled \hat{I}_p point from the middle graph of R to the middle graph of P. Blue arrows labeled \hat{I}_r point from the middle graph of P to the middle graph of R. A blue arrow labeled \hat{L}_r points from the middle graph of R to the right graph of R. A blue arrow labeled \hat{L}_p points from the middle graph of P to the right graph of P. A blue arrow labeled \hat{L}_r points from the middle graph of R to the right graph of P. A blue arrow labeled \hat{L}_p points from the middle graph of P to the right graph of R. A blue arrow labeled \hat{L}_r points from the middle graph of R to the right graph of R. A blue arrow labeled \hat{L}_p points from the middle graph of P to the right graph of P. A blue arrow labeled \hat{L}_r points from the middle graph of R to the right graph of P. A blue arrow labeled \hat{L}_p points from the middle graph of P to the right graph of R.

(=) (faithful trans)

$pre_R \rightarrow pre_P$: requestor must provide all information necessary for the execution of the provider operation,

$effect_P \rightarrow effect_R$: effect of the provided operation must include those expected by the requestor.

University of Leicester

Syntactic Matching: formally

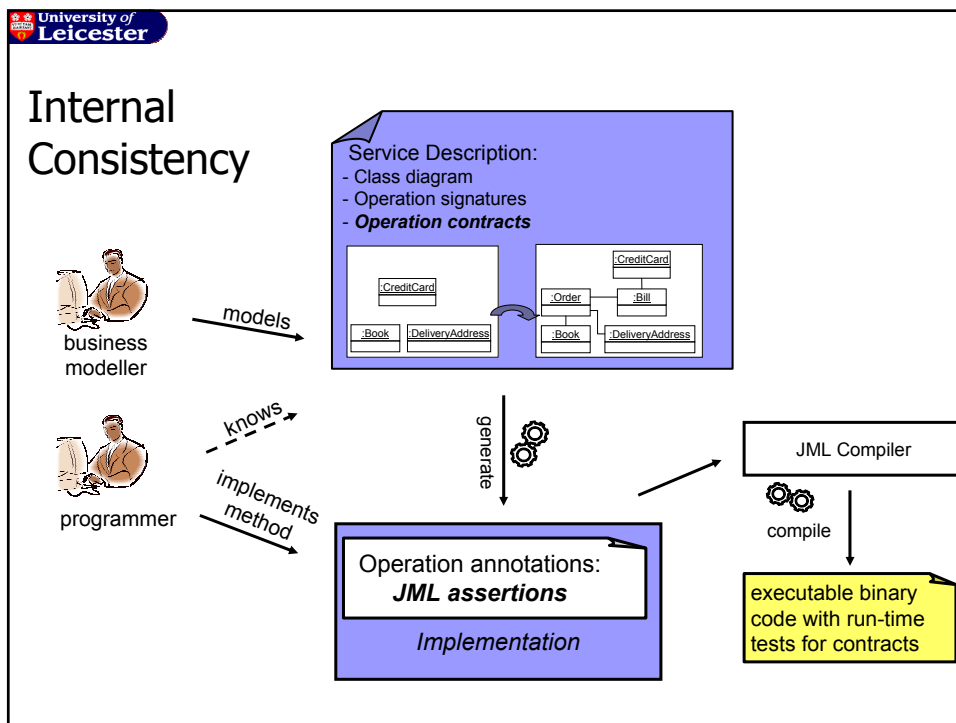
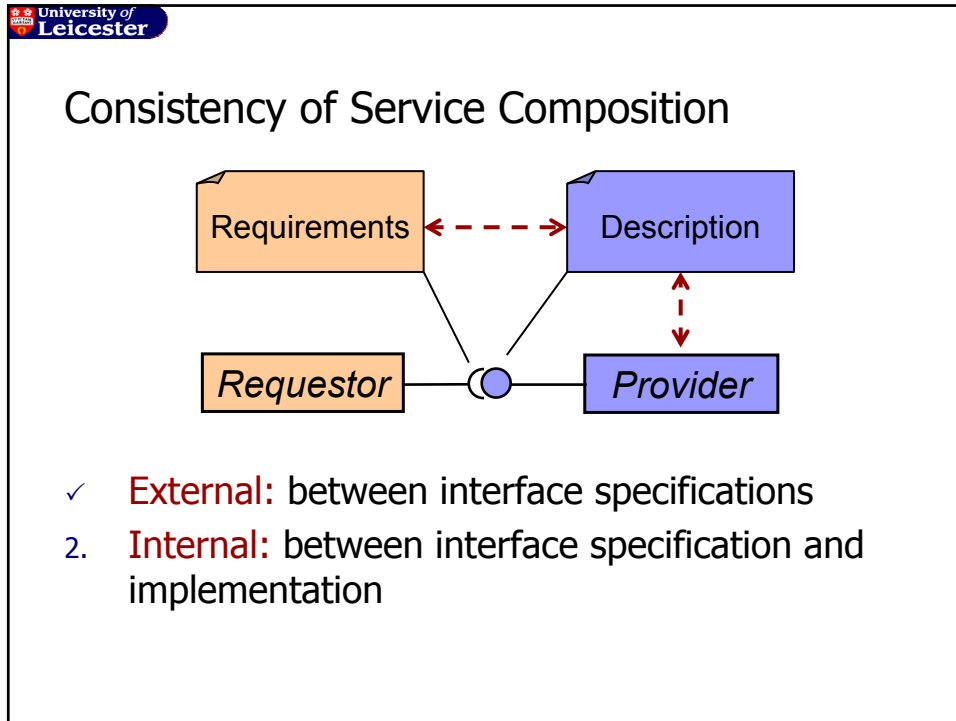
(p_1, \hat{L}_1) syntactically matches (p_2, \hat{L}_2) , in symbols $(p_2 : s_2, \hat{L}_2) \vdash_{match} (p_1 : s_1, \hat{L}_1)$, iff:

- there exists an injective graph homomorphism $h_{\hat{L}} : \hat{L}_2 \rightarrow \hat{L}_1$ s.t. $h_{\hat{L}} \circ \hat{l}_2$ satisfies *IC* of p_2 , and
- there exist graph homomorphisms $h_L : L_1 \rightarrow L_2$, $h_K : K_1 \rightarrow K_2$, and $h_R : R_1 \rightarrow R_2$ s.t. the diagrams (a), (b), and the diagram on the left commute, and the diagrams (a) and (b) represent a faithful transition.

University of Leicester

What do we have?

- ✓ Semantic compatibility: relation $| =$
- ✓ Syntactic matching: relation $| \dashv \dashv$
- ✓ Soundness: $p_2 | \dashv \dashv p_1$ implies $p_2 | = p_1$
- ✓ Completeness: $p_2 | = p_1$ implies $p_2 | \dashv \dashv p_1$



University of Leicester

JML from Graphical Contracts

Semantic idea: Assume rule r specifying method m .

- If r is applicable to G , then m invoked in G (with appropriate parameters) terminates without exception.
- If invocation yields H , there exists a graph transition from G to H via r .

After manually refining the models (business → analysis view), translate

1. class diagram → Java class frames
2. rules → JML
 - patterns
 - rules

University of Leicester

Class diagrams → Java class frames

```

classDiagram
    class ShopImplementation {
        <<control>>
        +orderProduct()
        +createOrder(in customerNo : Integer) : Integer
        +addProductToOrder(in productNo : Integer, in customerNo : Integer, in orderNo : Integer) : Boolean
        +clearOrder(in customerNo : Integer, in orderNo : Integer) : Boolean
        +saveOrder(in orderNo : Integer) : Boolean
    }
    class Product {
        <<entity>>
        +productNo : Integer
        +name : String
    }
    class Customer {
        <<entity>>
        +customerNo : Integer
        +name : String
    }
    class Order {
        <<entity>>
        +orderNo : Integer
        +creationDate : Date
    }
    ShopImplementation "1" -- "0..1" Product : productNo
    ShopImplementation "1" -- "0..1" Order : orderNo
    ShopImplementation "1" -- "0..1" Customer : customerNo
    Product "0..*" -- "0..*" Order : contains
    Order "0..*" o-- "0..*" Customer : buyer
    
```

UML attributes → private attributes with access methods

UML associations → pairs of attributes, mutually consistent

```

private int orderNo;
public int getOrderNo() {...}
public void setOrderNo(int no) {...}

private Customer buyer;
public void setBuyer(Customer c) {...}
public Customer getBuyer() {...}

private TreeSet revBuyer;
public void addRevBuyer(Order o){...}
public void removeRevBuyer(Order o){...}
public bool hasRevBuyer(Order o){...}
    
```

University of Leicester

Contracts → JML

```

public class ShopImplementation {
    ...
    /*      @ public normal_behavior
           @ requires JML-PRE;
           @ ensures JML-POST;
    */
    public boolean addProductToOrder(
        int productNo, int customerNo, int orderNo) {...}
    ...
}
    
```

UML Class Diagram:

- ShopImplementation** (Control):
 - Attributes: `pr : Product`, `o : Order`, `c : Customer`
 - Operations: `addProductToOrder(prNo, cNo, oNo):true`
- Product** (Entity):
 - Attributes: `productNo = pNo`, `name`
- Order** (Entity):
 - Attributes: `orderNo = oNo`, `creationDate`
- Customer** (Entity):
 - Attributes: `customerNo = cNo`, `name`

University of Leicester

Contracts → JML: Patterns

- starting at *this* navigate to as yet unbound objects, check attributes and links and bind them
- select navigation paths to achieve earliest possible failure

```

@ public normal_behavior
@ requires (\exists Product p;
@   product.contains(p);
@   p.getNo() == productNo);
@   && (\exists Customer c;
@     customer.contains(c);
@     c.getNo() == customerNo
@   && (\exists Order o;
@     order.contains(o);
@     o.getOrderNo() == orderNo
@     && o.getCustomer() == c
@     && o.containsProduct(p) == false));
    
```

University of Leicester

Contracts → JML: Rules

```

@ public normal_behavior
@
@ old Product p = getProductByNo(productNo);
@ old Customer c = getCustomerByNo(customerNo);
@ old Order o = c.findOrderByNo(orderNo);
@
@ requires p != null;
@ requires c != null;
@ requires o != null;
@ requires o.getCustomer() == c
@   && o.containsProduct(p) == false;
    
```

- like let, evaluated in pre state
- works for deterministic matches

```

@ ensures p != null;
@ ensures c != null;
@ ensures o != null;
@ ensures \not_modified(p, c);
@ ensures o.getCustomer() == c;
@ ensures o.getProducts().contains(p);
    
```

University of Leicester

Non-deterministic Matching

alternative class diagram

Solution: store all possible bindings and check that at least on satisfies post-condition

University of
Leicester

Consistency of Service Composition

Visual representation of contracts based on GT with loose semantics

- **External:** syntactic characterization of service compatibility
- **Internal:** mapping of contracts to JML

Open questions

- relation between business-level and analysis-level contracts
- verification of mapping GT → JML
- implementation and evaluation

University of
Leicester

Papers

With A. Cherchago, M. Lohmann: *A Formal Approach to Service Specification and Matching based on Conditional Graph Transformation*, ICGT 2004 in Rome

With M. Lohmann: *Model-Driven Development of Reactive Information Systems: From Graph Transformation Rules to JML Contracts*, to appear in STTT