Redesign of UML Class Diagrams: a formal Approach

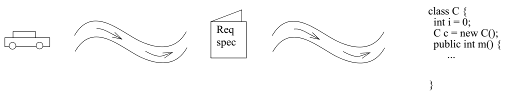
Piotr Kosiuczenko

Department of Computer Science
University of Leicester

Talk overview

- Problems with oo-approach
- Design by contract
- Interpretation Functions
- Examples
- Application to State Charts
- Requirements tracing

Software engineering



- OO-hype: code mirrors directly the real world
- Change is a constant factor in software development process: specification, design and implementation are not only being extended but can be changed
- A variety of design patterns is applied to generalize, improve decoupling, or optimize performance
- Continuous path from problem domain to code

Software engineering (UML)

- Structure:
  - Class diagram
- Interaction, Behaviour:
  - Object diagram
  - Sequence diagram
  - Collaboration
  - Statechart
- Requirements, Functionality:
  - Use Case Diagram
  - Activity Diagram
- Implementation:
  - Component Diagram
  - Deployment Diagram
- Object Constraint Language (OCL)

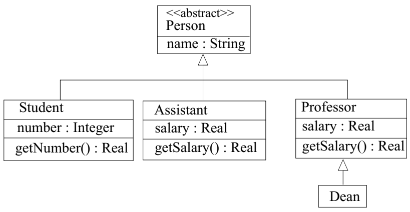
Design by Contract (R. Floyd/T. Hoare/B. Meyer)

- The Design by Contract approach allows us for system specification without getting into implementation details
- It separates what a class should do from how it should be done
- Contracts are formal specifications/agreements between the method caller and the method implementer
- A pre-condition specifies what should be true for the caller to make a request from the callee
- A post-condition specifies what should be true when the callee finishes completing the request
- An invariant specifies what should "always" be true

Object Constraints Language (OCL)

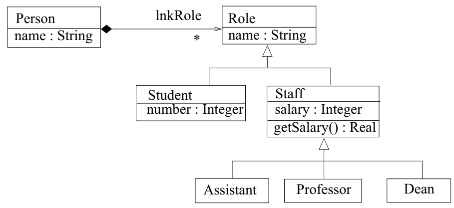
- OCL is a semiformal language for contractual specification of object-oriented systems
- It allows us to specify invariants as well as operation´s pre- and post-conditions

Example: role pattern




```
context Person inv personConstraint:
    not(self.oclIsTypeOf(Person))
           and
    self.oclIsKindOf(Dean) implies
                          self.oclIsKindOf(Professor)
```

Example cont.



Formal background (Hennicker et al.)

- `Boolean`, `String`, `Integer`, `Real` are modelled by the sorts Boolean, String, Integer, Real
- The object attributes are determined by <e, o>
  $<\_, \_> : Env \times C \longrightarrow Env\_C$
- Every OCL operation is modelled by a corresponding function
  $\_oclIsKindOf(\_) : Env\_Id \times ClN \longrightarrow Boolean$
- The attributes, associations and operations of a class C are modelled by corresponding functions on Env_C

Formal background (Hennicker at al.)

Trans : OCL ---> T(S, X)

Trans(`self`) $=_{df}$ self

Trans(`u.a`) $=_{df}$ <env, Trans(`u`)>.a

`self.`a_1`.` … `.`a_{n-1}`.`a_n ---> $a_n(env, a_{n-1}(env,...a_1(env, self)...))$

```
context C
inv : Ψ
```

$\forall_{env: Env,\ self : C,\ x1 : T1,\ \dots,\ xn : Tn}$ Trans(Ψ)

Basic notions

$A, B \subseteq T(\Sigma, X)$:

- A *generates* B iff A is contained in B and every non-variable term of B can be obtained from terms belonging to A by variable renaming and term composition
- The set A is a *base* of B iff in addition every term from B can be obtained in a unique way by composing terms from A and renaming of variables

Basic notions

Let $\varphi : T(S, F, \sigma, \leq, X) \rightarrow T(S', F', \sigma', \leq', X)$ be a partial function such that $var(\varphi(t)) \subseteq var(t)$.

φ is *compositional* iff for all terms t the following conditions hold:

- $\varphi(x) = x$, for $x \in X$
- if φ maps t_i on t_i', for $i = 0,\dots, n$, and t has the form $t_0[t_1/x_1,\dots, t_n/x_n]$, then φ is defined on t and $\varphi(t) = t_0'[t_1'/x_1,\dots, t_n'/x_n]$
- φ preserves predefined logical operators such as: $\wedge, \neg, \exists$.

Extendability theorem

Theorem
Let A, B $\subseteq$ T(S, F, $\leq$, σ, X) be sets of terms, B is closed on term composition and let $\psi : A \rightarrow T(S', F', \sigma', \leq', X)$. Let $\rho : S \rightarrow S'$ be a partial function. If the following conditions are satisfied:

- $var(\psi(t)) \subseteq var(t)$, if $\psi(t)$ is defined
- A is a base of B
- $\rho(\sigma(x)) \leq' \sigma'(x)$, for every variable $x \in X$
- $\sigma'(\psi(t)) \leq' \rho(\sigma(t))$, for every term $t \in A$
- $\sigma(t_1)) \leq \sigma(t_2)$ implies that $\rho(\sigma(t_1)) \leq' \rho(\sigma(t_2))$, for all terms $t_1, t_2 \in A$

Then ψ can be extended to B. Moreover ψ uniquely determined and compositional.

Orthogonal terms

A set of terms is *orthogonal*, if all terms in A are linear (no replications of variables) and A does not contain two terms u and v, such that:

- u is unifiable with a proper subterm of v or
- u is different from v, but u is unifiable with v.

Statement
If A is orthogonal, then A forms a basis of Gen(A).

Interpretation Functions

We call a compositional function ψ *interpretation function*, if and only if ψ is generated by a mapping with orthogonal domain.

Theorem
Let ψ be an interpretation function, and let E be an arbitrary set of first order formulas such that ψ is defined on them. If the formula Φ can be proved from E using

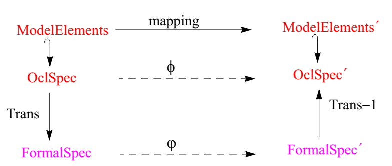
- equational reasoning (Birkhoff calculus),
- propositional tautologies,
- resolution rule and modus ponens,
- proof by induction, if the if the constructors in the range are images of constructors from the domain of ψ

then and $\psi(\Phi)$ logically follows from ψ(E).

Interpretation Functions

- The notion of interpretation function corresponds to the notion of refinement, but can deal with not incremental changes
- It corresponds also to the notion of abstraction in UML:
  a kind of dependency which relates two model elements that represent the same concept at different levels of abstraction or from different viewpoints

Transformation of OCL constraints



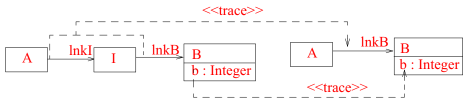
$\phi(x) =_{df} Trans^{-1}(\varphi(Trans(x))$

context C inv : Ψ context ϕ(C) inv : $\phi(\Psi)$

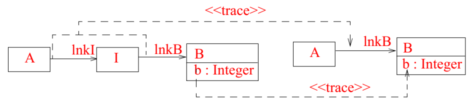
Examples

- Navigation path redesign
- Role pattern application
- State pattern application
- Refactoring

Navigation path redesign

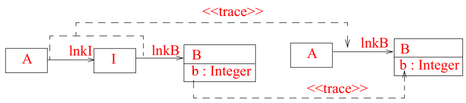


Navigation path redesign



- `A` ~> `A`
- `B` ~> `B`
- `lnkI.lnkB` ~> `lnkB`
- `b` ~> `b`

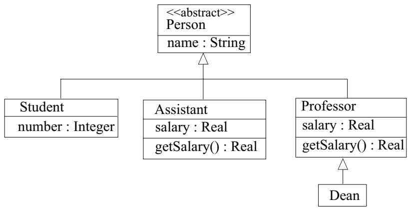
Navigation path redesign



- `A` ~> `A`
- `B` ~> `B`
- `lnkI.lnkB` ~> `lnkB`
- `b` ~> `b`

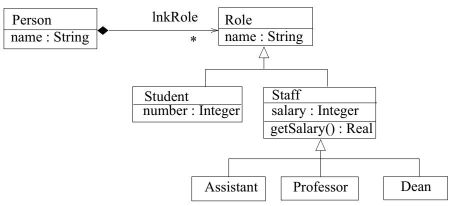
- <env, <env, self : A>.lnkI>.lnkB : B ~> <env, self : A>.lnkB : B
- <env, self : B>.b : Integer ~> <env, self : B>.b : Integer

Role pattern

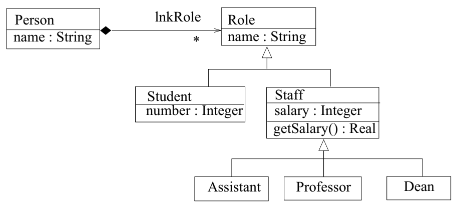



```
context Person inv personConstraint:
   not(self.oclIsTypeOf(Person))
         and
   (self.oclIsKindOf(Dean) implies
                    self.oclIsKindOf(Professor))
```

Example cont.

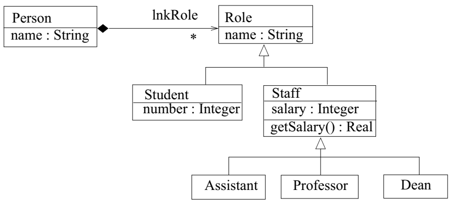


Example cont.



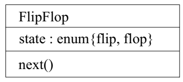

```
self.oclIsTypeOf(Person)      |----> self.lnkRole->isEmpty
self.oclIsKindOf(Professor)   |---->
      self.lnkRole->exists(r | r.oclIsKindOf(Professor))
self.oclIsKindOf(Dean)        |---->
      self.lnkRole->exists(r | r.oclIsKindOf(Dean))
```

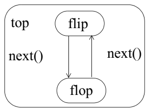
Example cont.



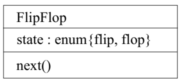

```
context Person inv personConstraint:
    not(self.lnkRole->isEmpty)
       and
    (self.lnkRole->exists(r | r.oclIsKindOf(Dean)) implies
       self.lnkRole->exists(r | r.oclIsKindOf(Professor)))
```

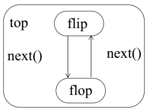
State Pattern





State Pattern





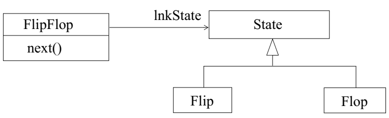

```
context FlipFlop :: next()post enumConstraint:
      state@pre = #flip implies state = #flop and
      state@pre = #flop implies state = #flip
```

Transforming pre- and post- conditions

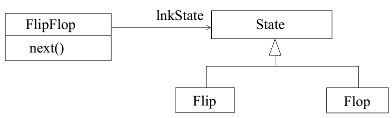
$$Trans_{post}(t.a) = \langle env', Trans_{post}(t)\rangle.a$$
$$Trans_{post}(t.a@pre) = \langle env, Trans_{post}(t)\rangle.a$$

$$\forall_{self : C,\ env,\ env' : Env,\ x1 : T1, \ldots, xn : Tn}\ [_{,\ result : T}\ result = \langle env, self\rangle.op(x_1, \ldots, x_n).T \wedge]\ env' = \langle env, self\rangle.op(x_1, \ldots, x_n).Env \wedge Trans(\Psi) \Rightarrow Trans_{post}(\Psi')$$

State pattern cont.

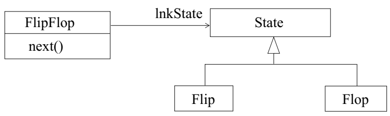


State pattern cont.




```
flip     ~>    Flip
flop     ~>    Flop
state    ~>    lnkState.oclType
```

State pattern cont.




```
flip     ~>    Flip
flop     ~>    Flop
state    ~>    lnkState.oclType

context FlipFlop :: next() post:
   lnkState@pre.oclType = Flip implies lnkState.oclType = Flop
       and
   lnkState@pre.oclType = Flop implies lnkState.oclType = Flip
```

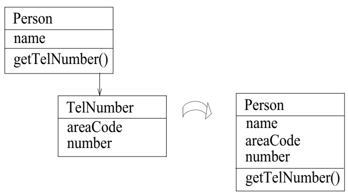
Refactoring Patterns (Fowler et. al.)

Refactoring is a technique for disciplined code redesign to make code clearer and cleaner.

Refectoring

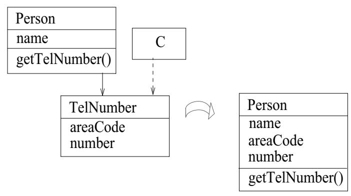
- applies refactoring patterns
- preserves functionality
- works in small steps
- after each step runnable (tested) code

Refactoring Patterns: Inline Class



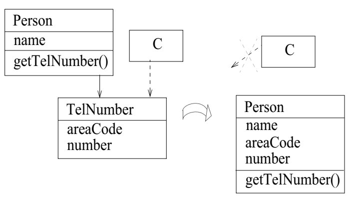

```
context Person inv:
   self.getTelNumber() = self.telNumber.number^context
context C inv:
   self.telNumber.number >= 100000
```

Refactoring Patterns: Inline Class

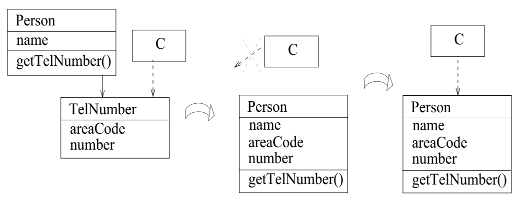



```
context Person inv:
   self.getTelNumber() = self.telNumber.number
context C inv:
   self.telNumber.number >= 100000
```

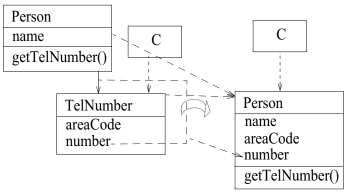
Refactoring Patterns: Inline Class



Refactoring Patterns: Inline Class

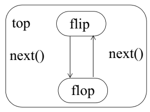


Refactoring Patterns: Inline Class



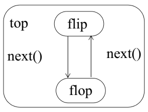

```
context Person inv:
self.getTelNumber() = self.number

context C inv:
self.person.number >= 100000
```

State Machines



- What happens to state machines when a transformation pattern is applied to the corresponding class diagrams?
- More precisely, what happens to states and their structural relations in a state machine?

State Machines



- State machines describes behavior of objects (model elements in general).
- A state in a state machine corresponds to a condition during the life of an object or an interaction during which it satisfies some condition, performs some action, or waits for some event.
- States can be specified by formulas, so called state invariants (e.g. OCL formulas).

SM Structure versus State Invariants

The topological relations between states can be interpreted by logical relations:

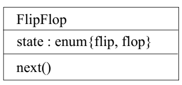
Let s_1 and s_2 be states and let I_1 and I_2 be the corresponding state invariants, respectively:

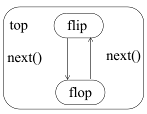
- Monotonicity: If s_1 is a substate of s_2, then I_1 implies I_2 (i.e. $I_1 \Rightarrow I_2$ holds).
- Non-overlappingness: If s_1 and s_2 are two different direct substates of an or-state, then I_1 excludes I_2 (i.e. $\neg(I_1 \wedge I_2)$ holds).

Let s be an or-state, let $s_1,..., s_n$ be all its substates and let $F, F_1,..., F_n$ be the corresponding formulas.

- Exhaustiveness: $F_1 \vee ... \vee F_n \Rightarrow F$

SM Structure versus State Invariants






```
top   –  self.state = flip or self.state = flop
flip  –  self.state = flip
flop  –  self.state = flop
```

Monotonicity:

self.state = flip ⇒ self.state = flip or self.state = flop
∧
self.state = flop ⇒ self.state = flip or self.state = flop

Non-overlappingness:

¬(self.state = flip ∧ self.state = flop)

Deriving logical invariants from UML metamodel

In general, the monotonicity condition can be expressed in OCL in respect to the UML metamodel as follows:
Let `smst` be the set of all states of a State Machine M (`smst` can be defined in OCL in a generic way). Then

```
smst.forAll(s.subvertex.forAll(v|
            v.constraint implies s.constraint))
```

The logical equivalent has the form:

$\wedge\{v.constraint \Rightarrow s.constraint \mid s \in smst \wedge v \in s.subvertex\}$

SM Structure as Logical Relation

In general, we consider propositional formulas of the form $C(y_1,\ldots, y_n)$.

Let A be a set of formulas. We say that states $s_1, \ldots, s_n$ satisfy the formula $C(y_1,\ldots, y_n)$ in respect to specification A and entailment relation $\vdash$ if and only if

$$A \vdash C(s_1,\ldots, s_n)$$

Let ψ map states $s_1, \ldots, s_n$ to states $s_1', \ldots, s_n'$. We say that ψ preserves propositional formula C in respect to A and to the entailment relation $\vdash$ if and only if

$A \vdash C(s_1,\ldots, s_n)$ implies that $A \vdash C(s_1',\ldots, s_n')$

Structural Invariants

Statement

Let $C(y_1,\ldots, y_n)$ be a propositional formula and let ψ be an interpretation function. Then, the following holds:

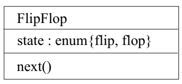
$$\psi(C(y_1,\ldots, y_n)) = C(\psi(y_1),\ldots, \psi(y_n)).$$

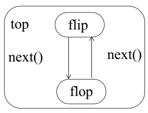
Corollary

Let A be a set of formulas. Let C be a propositional formula. Let ψ be defined on the formulas from A and on states s_j, for $j = 1,\ldots, n$. Then ψ preserves C in respect to A and $\vdash_{epri}$.

Consequently, IFs preserve structure of state machines as long as the relation between the corresponding invariants can be proved by above mentioned ways of reasoning.

Implementing States by Enumeration Types






```
top   –  self.state = flip or self.state = flop
flip  –  self.state = flip
flop  –  self.state = flop
```

We formally prove the non-overlappingness condition by contradiction using equational reasoning.

Proof of non-overlappingness

self.state = #flop $\wedge$ self.state = #flip implies #flop = self.state $\wedge$ self.state = #flip

because of $(a \Rightarrow b) \Rightarrow (c \wedge a) \Rightarrow (c \wedge b)$ and $x = y \Rightarrow y = x$

#flop = self.state $\wedge$ self.state = #flip implies #flop = #flip

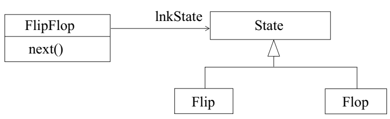
because of $x = y \wedge y = z \Rightarrow x = z$

$\neg$(self.state = #flop $\wedge$ self.state = #flip)

because #flip is different from #flop, $(p \Rightarrow q) \wedge (q \Rightarrow r) \Rightarrow (p \Rightarrow q)$, $(p \Rightarrow q) \wedge \neg q \Rightarrow \neg p$

Consequently, the states flip and flop are non-overlapping.

Example: State Pattern application



self.lnkState.oclType = Flop $\wedge$ self.lnkState.oclType = Flip implies

self.lnkState.oclType = Flop $\wedge$ self.lnkState.oclType = Flip

because of $(a \Rightarrow b) \Rightarrow (c \wedge a) \Rightarrow (c \wedge b)$ and $x = y \Rightarrow y = x$

Flop = self.lnkState.oclType $\wedge$ self.lnkState.oclType = Flip implies Flop = Flip

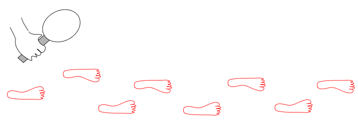
because of $x = y \wedge y = z \Rightarrow x = z$

$\neg$(self.lnkState.oclType = Flop $\wedge$ self.lnkState.oclType = Flip)

because Flip is different from Flop, $(p \Rightarrow q) \wedge (q \Rightarrow r) \Rightarrow (p \Rightarrow q)$, $(p \Rightarrow q) \wedge \neg q \Rightarrow \neg p$.

Consequently in the second case, the states flip and flop are non-overlapping as well.

Traceability



- You can't manage what you can't trace (R. Watkins, M. Neal)
- Tracing is usually practiced by software providers of high-reliability products and systems
- Requirements traceability is expressly demanded
  by the US Department of Defense and
  in the US health-care industry

Traceability

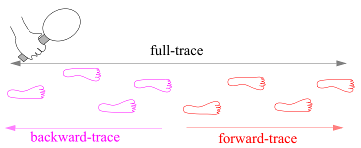
- Requirements traceability is the ability to describe and follow the life of a requirement, in both a forward and backward direction, throughout the system life cycle (M.. Jarke)
- Traceability allows us for (Catalysis):
  - providing a clear trace from requirements spec to implementation
  - justifying design decisions more clearly
  - clears the difference between requirements and their refinement
  - is a good cross-check, helping to expose inconsistencies
  - makes unambiguous statement about how the abstract model has been represented/implemented

Traceability

- Traceability should come as a side effect rather then impose additional bureaucracy
- Problems:

  Traceability is time consuming and error prone.
  Not much software support is offered.

Trace: Formal definition



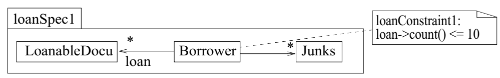
Let F_0 be a set of compositional functions

$F =_{df} \{(t, t') \mid \exists_{f \in F_0} f(t) = t'\}$

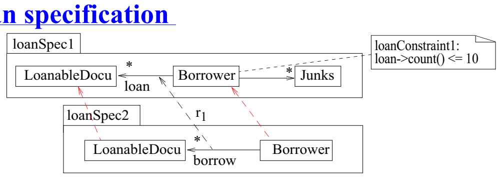
For a set of terms U

- the *forward-trace* of U equals $F^*(U)$
- the *backward-trace* of the set U equals $(F^*)^{-1}(U)$
- the *full-trace* of U equals $(F)^*(U) \cup (F^{-1})^*(U)$

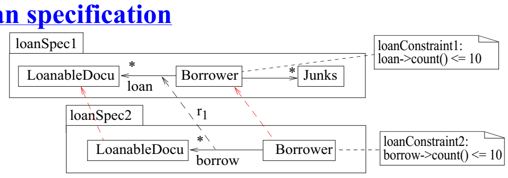
Trace: Loan specification



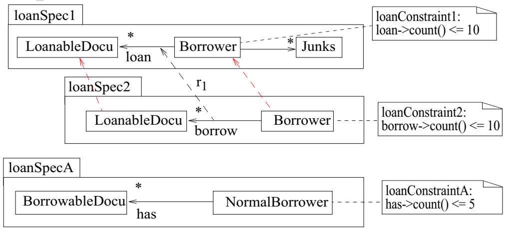
Trace: Loan specification



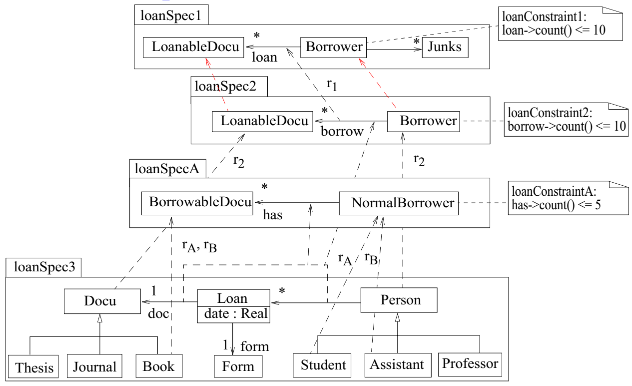
Trace: Loan specification



Trace: Loan specification



Trace: Loan specification



Trace: Loan specification

```
context loanSpec3::Person inv personConstraint :
   self.loan.doc->count() <= 10

context loanSpec3::Student inv studentConstraint :
   self.loan.doc->count() <= 5

context loanSpec3::Assistant inv assistantConstraint :
   self.loan.doc->count() <= 5
```

Trace: Loan specification

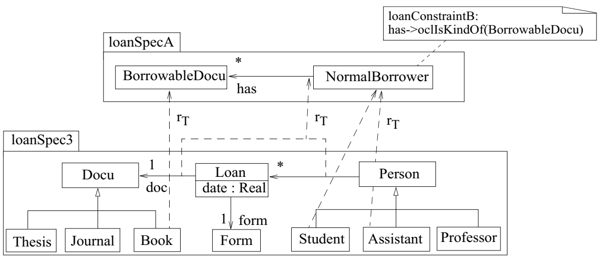
Forward-trace of `loanConstraint2`:
`personConstraint`

Backward-trace of `loanConstraint2`:
`loanConstraint1`

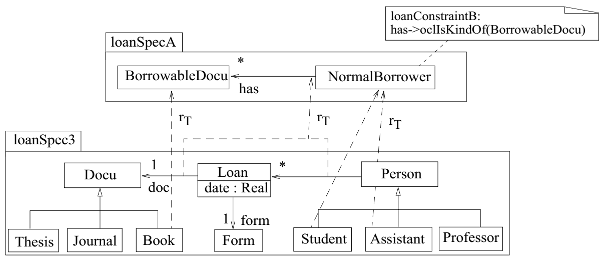
Full-trace of `loanConstraint2`:
`loanConstraint1`, `loanConstraint2`, `personConstraint`

Forward-trace of `loanConstraintA`:
`studentConstraint`, `assistantConstraint`

Trace: Loan specification



Trace: Loan specification




```
context loanSpec3::Student inv studentConstraintBD :
     self.loan.doc->oclIsKindOf(Book)

context loanSpec3::Assistant inv assistantConstraintBD :
     self.loan.doc->oclIsKindOf(Book)
```

Concluding remarks

Interpretation functions

- allow us for transformation of constraints
- preserve basic kinds of proofs
- preserve structure of state machines
- preserve LTL/CTL proofs
- can be implemented, but are syntax sensitive

Future work

- Tool support
- Proof of the conjecture that IFs preserve first order logicentailment relation (counterexample, resp.)
- Relation to institutions
- Integration with software development methods such as Catalysis and RUP
- Definition of interpretation functions modulo equational theory