Introduction to OCL

Wintersemester 2010/2011

Prof. Dr. Jan Jürjens and Dr.-Ing. Holger Schmidt

TU Dortmund – Department of Computer Science
Software Engineering (LS 14)
http://ls14-www.cs.tu-dortmund.de/

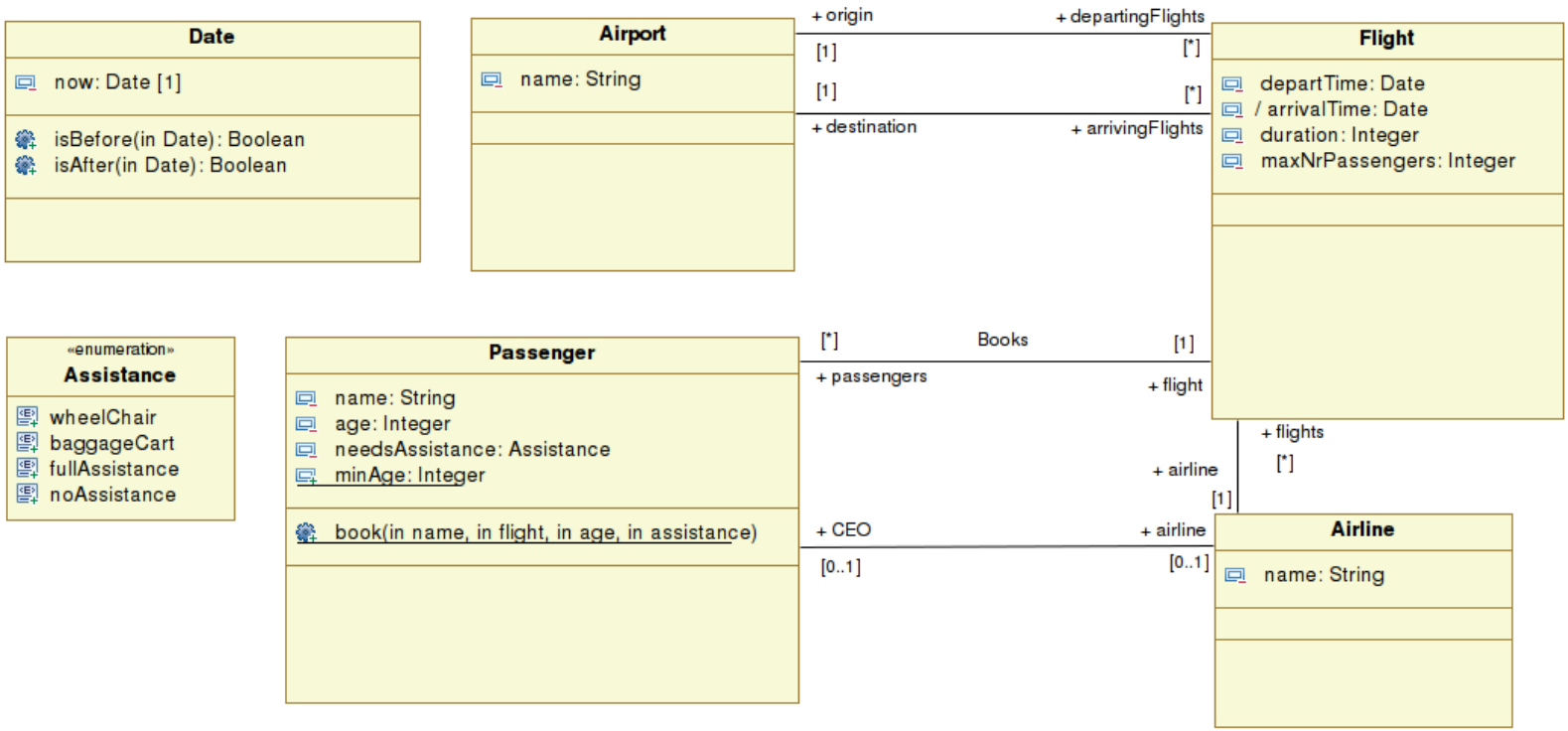
Slides are based on the lecture "Muster- und Komponenten-basierte Softwareentwicklung" by Prof. Dr. Maritta Heisel

Motivation for Formal Model-Based Specification

- UML (Unified Modeling Language) 2.0 [UML09] is a (semi-formal) modeling language proposed by the OMG (Object Management Group)[1].
- UML is the de facto industry standard notation to model software analysis and design artifacts.
- UML Superstructure specification 2.2 [2] describes 14 (semi-)formal diagram types, e.g., class and use-case diagrams.
- Limits:
  - not precise and automatic verification hardly possible
  - weak code generation capabilities (usually only code skeletons, not fully functional code)

[1] http://www.omg.org/
[2] http://www.omg.org/spec/UML/2.2/

Running Example: Airport Class Diagram



- How many passengers can be registered for a flight?

Formal Models

- (Semi-formal) visual models can be enriched with formal specifications of
  - state constraints (with invariants)
  - operation semantics (with pre- and post-conditions)
- UML defines a language that can be used with this goal: Object Constraint Language (OCL)
- Advantages:
  - UML diagrams enriched with OCL expressions lead to precise specifications that can be verified automatically
  - formal specifications remove the ambiguity that characterizes informal specifications
  - formal specifications can be automatically verified
  - tools exist that generate code and assertions in Java from OCL specifications of state invariants and operations' pre- and post conditions

What is OCL?

- OCL is a formal language used to describe constraints on UML models.
- OCL is not a programming language; therefore, it is not possible to write program logic or flow control in OCL.
- OCL expressions are guaranteed to be without side effects:
  - when an OCL expression is evaluated, it simply returns a value; it cannot change anything in the model
  - the state of the system will never change because of the evaluation of an OCL expression, even though an OCL expression can be used to specify a state change (e.g., in a post-condition)
- OCL supports strong type checking.

Literature

OMG specification:

- "Object Constraint Language 2.0" [UML10]
  `http://www.omg.org/spec/OCL/2.2/PDF`

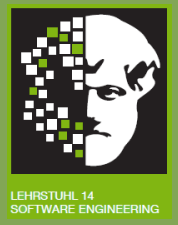
(Partly) basis for our "Introduction to OCL":

- "The Object Constraint Language: Getting Your Models Ready for MDA" [WK03]
- "OCL – Object Constraint Language" (slides)
  `http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf`

Specification of OCL Expressions

OCL expressions

- are always bound to a UML model
- always put constraints on the elements of the UML model they belong to; this model describes which classes may be used and which attributes, operations, and associations are available for objects from these classes
- are denoted in the UML model they belong to or in a separate document



Definitions

Constraint A restriction on one or more parts of a UML model.

Class invariant A constraint that must (almost) always be met by all instances of a class.

Pre-condition A constraint that must be true before the execution of an operation.

Post-condition A constraint that must be true after the execution of an operation.

Guard condition A constraint that must be true before a transition in a statechart/state diagram, or analogously a message in a sequence diagram, and other behavioral UML diagrams.

Basic Format of an OCL Expression

context <identifier> <constraintType>
[<constraintName>]:<boolean expression>

context a keyword to mark the relative model element indicated by <identifier> from which other model elements can be referenced. The keyword *self* can be used within <boolean expression> to access the *context*.

<identifier> is a class or operation name

<constraintType> is one of the keywords *inv*, *pre*, or *post*

<constraintName> is an optional name for the constraint

<boolean expression> is some boolean expression, often an equation

OCL Types and Keywords

The following types can be used in an OCL expression:

- predefined types
  - primitive types: String, Integer, Real, Boolean
  - collection types: Set, Bag, Sequence, OrderedSet
  - tuple types: Tuple
  - special types: OclType, OclAny, ...
- classifiers from the UML model and their features
  - classes, enumeration classes, and role names
  - attributes and operations

The following keywords can be used in an OCL expression:

- *if* – *then* – *else* – *endif*: conditional expression
- *not*, *or*, *and*, *xor*, *implies* boolean operators
- *def* global definitions
- *let* – *in* local definitions

Class Invariants

- An invariant
  - is a condition that must hold before and after execution of a method, but can be violated during method execution
  - is specified with the keyword *inv* in the context of an instance of a classifier (class, role name, . . .)

Common types of invariants

- Domain constraints:
  constraints on the set of possible values of an attribute
- Unique constraints:
  an attribute or set of attributes in a class that cannot take the same value or set of values for two distinct instances of the class
- Time constraints
- Constraints that define derived model elements (e.g., derived attributes)
- Existence rules:
  rules that state that certain objects/values should exist/be defined when other objects/values exist/are defined

Invariants on Attributes I

- The class to which the invariant refers is the context of the invariant.
- It is followed by a boolean expression that states the invariant.
- All attributes of the context class may be used in this invariant.

Example

context Flight
inv : *duration* < 4

- Meaning: ?

Invariants on Attributes I

- The class to which the invariant refers is the context of the invariant.
- It is followed by a boolean expression that states the invariant.
- All attributes of the context class may be used in this invariant.

Example

context Flight
inv : *duration* < 4

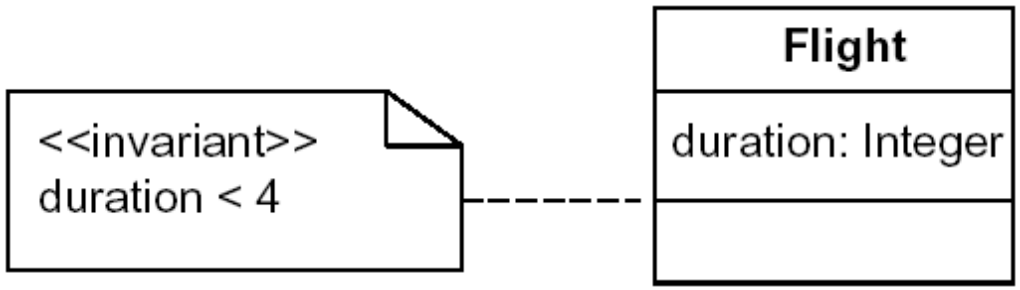
- Meaning:
  - Each flight has a duration of less than 4h.

Usage of self

The following invariant notations are equivalent:

context Flight
inv : self.duration < 4

context Flight
inv : duration < 4



Invariants on Attributes II

- If the type of the attribute is a class, the attributes or query operations defined on that class can be used to write the invariant (using a dot notation).
- Query operation:
  An operation that does not change the value of any attributes.

Example

context Flight
inv : *departTime.isBefore*(*arrivalTime*)

- Meaning: ?

Invariants on Attributes II

- If the type of the attribute is a class, the attributes or query operations defined on that class can be used to write the invariant (using a dot notation).
- Query operation:
  An operation that does not change the value of any attributes.

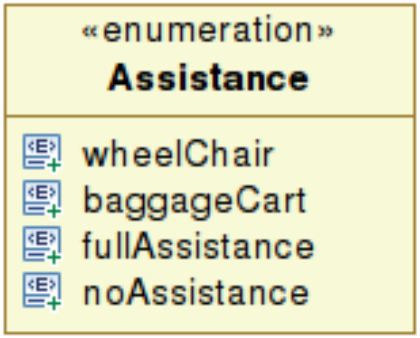
Example

context Flight
inv : *departTime.isBefore*(*arrivalTime*)

- Meaning:
  - The departure date is earlier than the arrival date.

Enumeration Types

- Enumeration uses datatype followed by :: and the value



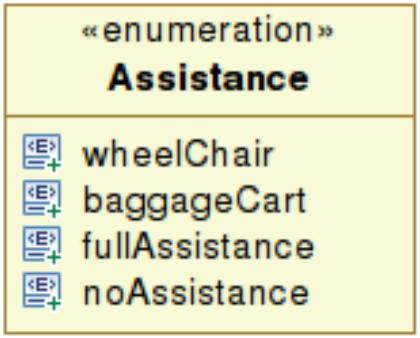
Example

context Passenger
inv : self.age $>$ 95 *implies*
self.needsAssistance $=$ *Assistance :: wheelchair*

- Meaning: ?

Enumeration Types

- Enumeration uses datatype followed by :: and the value



Example

context Passenger
inv : self.age $>$ 95 *implies*
self.needsAssistance $=$ *Assistance :: wheelchair*

- Meaning:
  - Each passenger with an age above 95 needs assistence by a wheelchair.

Associations and Navigation I

- Every association is a navigation path.
- The context of the expression is the starting point.
- Role names (or association ends) are used to identify the navigated associations.

Associations and Navigation II

Example

context Flight
inv : origin <> destination

- Meaning: ?

Example

context Flight
inv : origin.name = 'Duisburg'

- Meaning: ?

Associations and Navigation II

Example

context Flight
inv : origin <> destination

- Meaning:
  - The origin of each flight is unequal to the destination.

Example

context Flight
inv : origin.name = 'Duisburg'

- Meaning: ?

Associations and Navigation II

Example

context Flight
inv : origin <> destination

- Meaning:
  - The origin of each flight is unequal to the destination.

Example

context Flight
inv : origin.name = 'Duisburg'

- Meaning:
  - The origin of each flight is Duisburg.

Associations and Navigation III

- Often associations are one-to-many or many-to-many, which means that constraints on a collection of objects are necessary.
- OCL expressions either state a fact about all objects in the collection or states facts about the collection itself.

Using Collection Operations I

- One of the collection operations can be used whenever navigation results in a collection of objects.
- An arrow ($->$) between the rolename and the operation indicates the use of one of the predefined collection operations (e.g. $passengers->size()$).
- A dot (.) between the rolename and the operation indicates the use of one of an operation defined in the UML model (e.g. $departTime.isBefore(arrivalTime)$).

Example

context Flight
inv : passengers->size() <= maxNrPassengers

- Meaning: ?

Using Collection Operations I

- One of the collection operations can be used whenever navigation results in a collection of objects.
- An arrow ($->$) between the rolename and the operation indicates the use of one of the predefined collection operations (e.g. $passengers->size()$).
- A dot (.) between the rolename and the operation indicates the use of one of an operation defined in the UML model (e.g. $departTime.isBefore(arrivalTime)$).

Example

context Flight
inv : passengers->size() <= maxNrPassengers

- Meaning:
  - The number of passengers is less or equal to the maximum number of seats.

Using Collection Operations II

A collection of objects may be:

- Set:
  - Each element may occur only once.
  - Single navigation of an association results in a Set.
- Bag:
  - Elements may be present more than once.
  - Combined navigation results in a Bag.
- OrderedSet:
  - A set in which the elements are ordered.
  - Single navigation of an association that is marked as $\{ordered\}$ results in an OrderedSet.
- Sequence:
  - A Bag in which the elements are ordered.
  - Combined navigation of associations, at least one of which is marked as $\{ordered\}$, results in an Sequence.

The *collect* Operation I

- The operation can be used to collect attribute values, e.g. $passengers->collect(name)$.

Meaning (in pseudo code)

```
Collection<String> c = new Collection();
foreach (p:  passengers) {c.add(p.name);}
return c;
```

- The operation also can be used to build a new collection from the objects held by association ends, e.g. $arrivingFlights->collect(airline)$.

Meaning (in pseudo code)

```
Collection<Airline> c = new Collection();
foreach (f:  arrivingFlights) {c.add(f.airline);}
return c;
```

The *collect* Operation II

- The resulting collection contains different objects from the original collection.
- When the source collection is a Set the resulting collection is not a Set but a Bag.
- If the source collection is a Sequence or an OrderedSet, the resulting collection is a Sequence.
- The dot notation is an abbreviation for applying the *collect* operation:
  - *passengers.name*
  - *arrivingFlights.airline*

The *collect* Operation III

Example

$$context\ Airport$$
$$inv\ :\ arrivingFlights->size() = arrivingFlights->collect(airline)->size()$$

- Meaning: ?

The *collect* Operation III

Example

$$context\ Airport$$
$$inv\ :\ arrivingFlights->size() = arrivingFlights->collect(airline)->size()$$

- Meaning:
  - Each arriving flight is carried out by an airline.

The *select* Operation

- The *select* operation takes an OCL expression as parameter.
- The result of *select* is a subcollection of the collection on which it is applied.
- *select* selects all elements from the collection for which the expression evaluates to true.

Example

$$\begin{aligned} &\textit{context Flight} \\ &\textit{inv} : \textit{passengers}->\textit{select}(\textit{needsAssistance} <> \\ &\quad \textit{Assistance} :: \textit{noAssistance})->\textit{size}() <= 10 \end{aligned}$$

- Meaning: ?

The *select* Operation

- The *select* operation takes an OCL expression as parameter.
- The result of *select* is a subcollection of the collection on which it is applied.
- *select* selects all elements from the collection for which the expression evaluates to true.

Example

context Flight
inv : passengers$->$ *select*(*needsAssistance* $<>$ *Assistance* :: *noAssistance*)$->$ *size*() $<=$ 10

- Meaning:
  - The number of passengers who need assistance is less or equal to 10.

The *reject* Operation

- The *reject* operation is analogous to *select*.
- *reject* selects all elements from the collection for which the expression evaluates to false.

Example

context Flight

$$inv : passengers->reject(needsAssistance = Assistance::noAssistance)->size() <= 10$$

- Meaning: **?**

The *reject* Operation

- The *reject* operation is analogous to *select*.
- *reject* selects all elements from the collection for which the expression evaluates to false.

Example

context Flight

$$inv : passengers -> reject(needsAssistance = Assistance :: noAssistance) -> size() <= 10$$

- Meaning:
  - The number of passengers who need assistance is less or equal to 10.

The *forAll* Operation I

- The *forAll* operation can be used to specify that a certain condition must hold for all elements of a collection.
- The *forAll* operation takes an OCL expression as parameter.
- This operation is used when there already is a (sub)set of all instances of a class, and the elements of of that (sub)set should be checked.
- The result of the operation is a boolean value:
  - true if the expression evaluates to true for all elements in the collection
  - otherwise false

The *forAll* Operation II

- *class.allInstances*(): collection of all instances of the class

Example

context Airport
inv : Airport.allInstances()−> *forAll*(a1, a2 |
a1 <> a2 *implies* a1.*name* <> a2.*name*)

- Meaning: **?**

The *forAll* Operation II

- *class.allInstances*(): collection of all instances of the class

Example

context Airport
inv : Airport.allInstances()−> *forAll*(*a*1, *a*2 |
*a*1 <> *a*2 *implies a*1.*name* <> *a*2.*name*)

- Meaning:
  - Each airport name is unique.
- Equivalent:

context Airport
inv : Airport.allInstances()−> *isUnique*(*name*)

Pre- and Post-Conditions I

- In class diagrams only the syntax and signature of operations can be defined.
- Operation semantics can be specified through pre- and post-conditions in OCL.
- Pre-condition:
  - condition on the arguments and initial object state that must hold for the operation call to be valid

Example for Pre-Condition

Example

$$context\ Passenger :: book(name : String, flight : Flight, age : Integer, assistance : Assistance)$$
$$pre : flight.passengers->size() < flight.maxNrPassengers$$

- Meaning: **?**

Example for Pre-Condition

Example

$$context\ Passenger :: book(name : String, flight : Flight, age : Integer, assistance : Assistance)$$
$$pre : flight.passengers->size() < flight.maxNrPassengers$$

- Meaning:
  - The amount of passengers registered for *flight* before the execution of *book* must be less than *maxNrPassengers*.

Pre- and Post-Conditions II

- Post-condition:
  - condition on the return value, final object state, arguments, and initial object state that must hold in the end of the operation execution, assuming the pre-condition is satisfied
  - specifies intended result and state change (what), but not the steps (how)
  - the pre state of an object field is indicated with @*pre*
  - the returned value is indicated with the keyword *result*

Example for Post-Condition

Example

$$
\begin{aligned}
&\mathit{context}\ \mathit{Passenger} :: \mathit{book}(\mathit{name} : \mathit{String},\ \mathit{flight} : \mathit{Flight},\\
&\qquad \mathit{age} : \mathit{Integer},\ \mathit{assistance} : \mathit{Assistance})\\
&\mathit{post}\ :\ \mathit{flight.passengers}->\mathit{size}()-\\
&\qquad \mathit{flight.passengers@pre}->\mathit{size}() = 1\ \mathit{and}\\
&\qquad \mathit{flight.passengers}->\mathit{exists}(p : \mathit{Passenger} \mid p.\mathit{age} = \mathit{age}\\
&\qquad \mathit{and}\ p.\mathit{name} = \mathit{name}\\
&\qquad \mathit{and}\ p.\mathit{needsAssistance} = \mathit{assistance})
\end{aligned}
$$

- Meaning: ?

Example for Post-Condition

Example

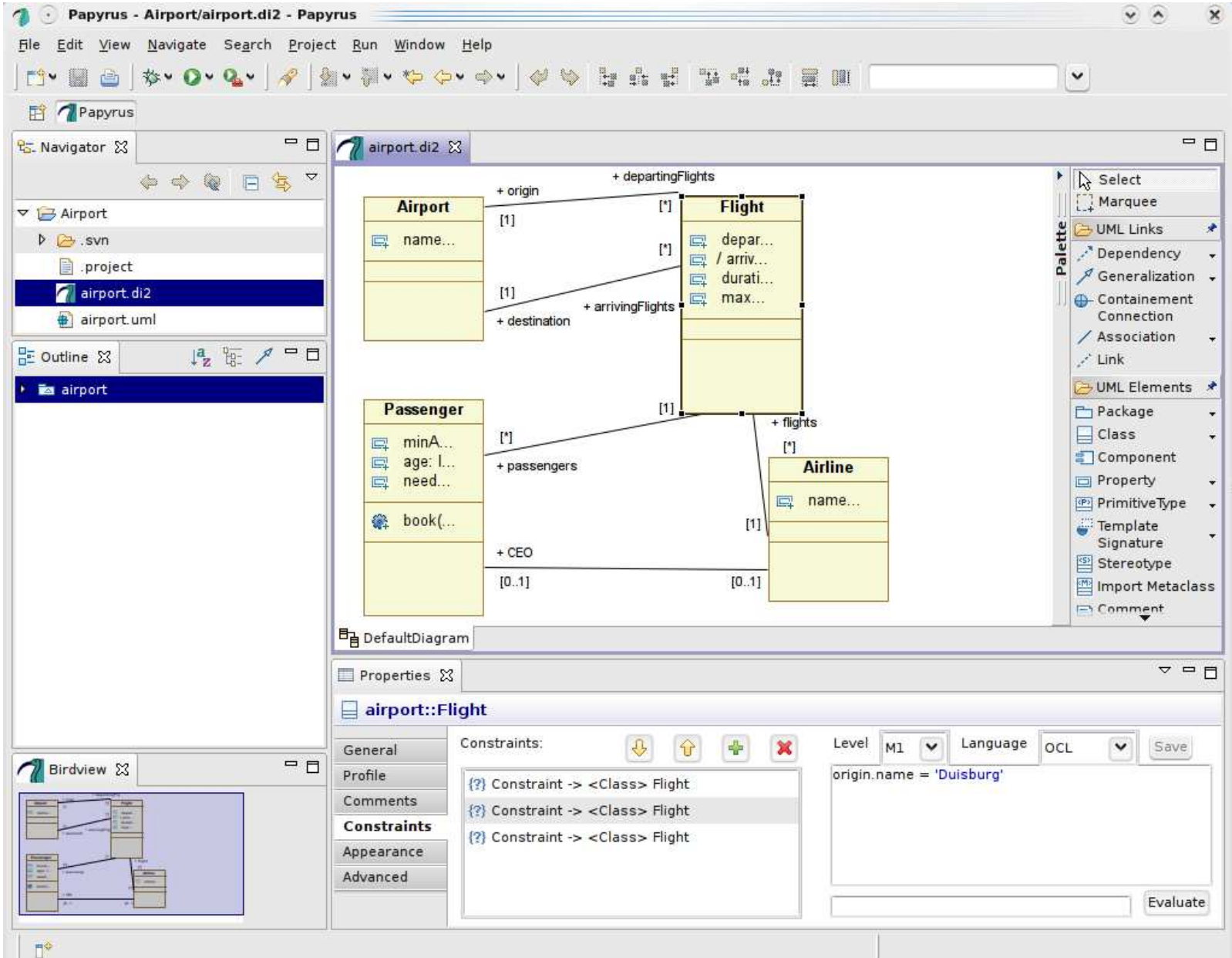
$$
\begin{aligned}
&\mathit{context}\ \mathit{Passenger}::\mathit{book}(\mathit{name}:\ \mathit{String},\ \mathit{flight}:\ \mathit{Flight},\\
&\qquad \mathit{age}:\ \mathit{Integer},\ \mathit{assistance}:\ \mathit{Assistance})\\
&\mathit{post}\ :\ \mathit{flight.passengers}->\mathit{size}()-\\
&\qquad \mathit{flight.passengers@pre}->\mathit{size}()=1\ \mathit{and}\\
&\qquad \mathit{flight.passengers}->\mathit{exists}(p:\mathit{Passenger}|p.\mathit{age}=\mathit{age}\\
&\qquad \mathit{and}\ p.\mathit{name}=\mathit{name}\\
&\qquad \mathit{and}\ p.\mathit{needsAssistance}=\mathit{assistance})
\end{aligned}
$$

- Meaning:
  - one additional object exists after execution
  - the attributes of one object have been initialized using the parameter values of *book*

Papyrus I

- Papyrus is a free open-source tool for modelling with UML 2.0.
- Download: `http://www.papyrusuml.org/`
- Based on the Eclipse environment.
- Full respect of the UML 2.0 standard as defined by the OMG (according to website).
- Extendable architecture of Papyrus that allows users to add new diagrams, new code generators, etc.
- Allows OCL constraints to be embedded in models.

Papyrus II



Primitive Types

Taken from http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf

Type	Description	Values	Operators and Operations
Boolean		true, false	=, <>, and, or, xor, not, implies, if-then-else-endif (note 2)
Integer	A whole number of any size	-1, 0, 1, ...	=, <>, >, <, >=, <=, *, +, - (unary), - (binary), / (real), abs(), max(b), min(b), mod(b), div(b)
Real	A real number of any size	1.5, ...	=, <>, >, <, >=, <=, *, +, - (unary), - (binary), /, abs(), max(b), min(b), round(), floor()
String	A string of characters	'a', 'John'	=, <>, size(), concat(s2), substring(lower, upper) (1<=lower<=upper<=size), toReal(), toInteger()

Notes:
1) Operations indicated with parenthesis are applied with ".", but the parenthesis may be omitted.
2) Example: title = (if isMale then 'Mr.' else 'Ms.' endif)

Collections and Tuples

Taken from `http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf`

Description	Syntax	Examples
Abstract collection of elements of type T	**Collection(T)**	
Unordered collection, no duplicates	**Set(T)**	Set{1 , 2}
Ordered collection, duplicates allowed	**Sequence(T)**	Sequence {1, 2, 1} Sequence {1..4} (same as {1,2,3,4})
Ordered collection, no duplicates	**OrderedSet(T)**	OrderedSet {2, 1}
Unordered collection, duplicates allowed	**Bag(T)**	Bag {1, 1, 2}
Tuple (with named parts)	**Tuple(field1: T1, … fieldn : Tn)**	Tuple {age: Integer = 5, name: String = 'Joe' } Tuple {name = 'Joe', age = 5}

Note 1: They are *value types*: "=" and "<>" compare values and not references.

Note 2: Tuple components can be accessed with "." as in "t1.name"

Operations on Collection(T)

Taken from http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf

Operation	Description
size(): Integer	The number of elements in this collection (*self*)
isEmpty(): Boolean	size = 0
notEmpty(): Boolean	size > 0
includes(object: T): Boolean	True if *object* is an element of *self*
excludes(object: T): Boolean	True if *object* is not an element of *self*
count(object: T): Integer	The number of occurrences of *object* in *self*
includesAll(c2: Collection(T)): Boolean	True if *self* contains all the elements of *c2*
excludesAll(c2: Collection(T)): Boolean	True if *self* contains none of the elements of *c2*
sum(): T	The addition of all elements in *self* (T must support "+")
product(c2: Collection(T2)) : Set(Tuple(first:T, second:T2))	The cartesian product operation of *self* and *c2*.

Note: Operations on collections are applied with "->" and not "."

Iterator Expressions on Collection(T) I

Taken from http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf

Iterator expression	Description
iterate(iterator: T; accum: T2 = init | body) : T2	Returns the final value of an accumulator that, after initialization, is updated with the value of the *body* expression for every element in the *source* collection.
exists(iterators | body) : Boolean	True if *body* evaluates to true for at least one element in the *source* collection. Allows multiple iterator variables.
forAll(iterators | body): Boolean	True if *body* evaluates to true for each element in the source collection. Allows multiple iterator variables.
one(iterator| body): Boolean	True if there is exactly one element in the *source* collection for which *body* is true
isUnique(iterator| body): Boolean	Results in true if *body* evaluates to a different value for each element in the *source* collection.
any(iterator | body): T	Returns any element in the source collection for which *body* evaluates to true. The result is null if there is none.
collect(iterator| body): Collection(T2)	The Collection of elements resulting from applying *body* to every member of the *source* set.

Note: The iterator variable declaration can be omitted when there is no ambiguity.

Iterator Expressions on Collection(T) II

Taken from http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf

Iterator expression	Description
select(iterator | body): Collection(T)	The Collection of elements of the *source* collection for which *body* is true. The result collection is of the same type of the *source* collection.
reject(iterator | body): Collection(T)	The Collection of elements of the *source* collection for which *body* is false. The result collection is of the same type of the *source* collection.
collectNested(iterator | body): CollectionWithDuplicates(T2)	The Collection of elements (allowing duplicates) that results from applying *body* (of type T2) to every member of the *source* collection. The result is not flattened. Conversions: Set -> Bag, OrderedSet -> Sequence.
sortedBy(iterator | body): OrderedCollection(T)	Returns an ordered Collection of all the elements of the *source* collection by ascending order of the value of the *body* expression. The type T2 of the *body* expression must support “<”. Conversions: Set -> OrderedSet, Bag -> Sequence.

Operations on Set(T) I

Taken from `http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf`

Operation	Description
=(s: Set(T)) : Boolean	Do *self* and *s* contain the same elements?
union(s: Set(T)): Set(T)	The union of *self* and *s*.
union(b: Bag(T)): Bag(T)	The union of *self* and bag *b*.
intersection(s: Set(T)): Set(T)	The intersection of *self* and *s*.
intersection(b: Bag(T)): Set(T)	The intersection of *self* and *b*.
-(s: Set(T)) : Set(T)	The elements of *self*, which are not in *s*.
including(object: T): Set(T)	The set containing all elements of *self* plus *object*.
excluding(object: T): Set(T)	The set containing all elements of *self* minus *object*.
symmetricDifference(s: Set(T)): Set(T)	The set containing all the elements that are in *self* or *s*, but not in both.

Operations on Set(T) II

Taken from `http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf`

Operation	Description
flatten() : Set(T2)	If T is a collection type, the result is the set with all the elements of all the elements of *self*; otherwise, the result is *self*.
asOrderedSet(): OrderedSet(T)	OrderedSet with elements from *self* in undefined order.
asSequence(): Sequence(T)	Sequence with elements from *self* in undefined order.
asBag(): Bag(T)	Bag will all the elements from *self*.

Operations on Bag(T)

Taken from http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf

Operation	Description
=(bag: Bag(T)) : Boolean	True if *self* and *bag* contain the same elements, the same number of times.
union(bag: Bag(T)): Bag(T)	The union of *self* and *bag*.
union(set: Set(T)): Bag(T)	The union of *self* and *set*.
intersection(bag:Bag(T)): Bag(T)	The intersection of *self* and *bag*.
intersection(set: Set(T)): Set(T)	The intersection of *self* and *set*.
including(object: T): Bag(T)	The bag with all elements of *self* plus *object*.
excluding(object: T): Bag(T)	The bag with all elements of *self* without *object*.
flatten() : Bag(T2)	If T is a collection type: bag with all the elements of all the elements of *self*; otherwise: *self*.
asSequence(): Sequence(T)	Seq. with elements from *self* in undefined order.
asSet(): Set(T)	Set with elements from *self*, without duplicates.
asOrderedSet(): OrderedSet(T)	OrderedSet with elements from *self* in undefined order, without duplicates.

Operations on Sequence(T) I

Taken from http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf

Operation	Description
=(s: Sequence(T)) : Boolean	True if *self* contains the same elements as *s*, in the same order.
union(s: Sequence(T)): Sequence(T)	The sequence consisting of all elements in *self*, followed by all elements in *s*.
flatten() : Sequence(T2)	If T is a collection type, the result is the set with all the elements of all the elements of *self*; otherwise, it's *self*.
append(object: T): Sequence(T)	The sequence with all elements of *self*, followed by *object*.
prepend(obj: T): Sequence(T)	The sequence with *object*, followed by all elements in *self*.
insertAt(index : Integer, object : T) : Sequence(T)	The sequence consisting of *self* with *object* inserted at position *index* (1<=index<=size+1)
subSequence(lower : Integer, upper: Integer) : Sequence(T)	The sub-sequence of *self* starting at index *lower*, up to and including index *upper* (1<=lower<=upper<=size)

Operations on Sequence(T) II

Taken from http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf

Operation	Description
at(i : Integer) : T	The *i*-th element of *self* (1<=i<=size)
indexOf(object : T) : Integer	The index of *object* in *self*.
first() : T	The first element in *self*.
last() : T	The last element in *self*.
including(object: T): Sequence(T)	The sequence containing all elements of *self* plus *object* added as last element
excluding(object: T): Sequence(T)	The sequence containing all elements of *self* apart from all occurrences of *object*.
asBag(): Bag(T)	The Bag containing all the elements from *self*, including duplicates.
asSet(): Set(T)	The Set containing all the elements from *self*, with duplicates removed.
asOrderedSet(): OrderedSet(T)	An OrderedSet that contains all the elements from *self*, in the same order, with duplicates removed.

Operations on OrderedSet(T)

Taken from http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf

Operation	Description
append(object: T): OrderedSet(T)	The set of elements, consisting of all elements of *self*, followed by *object*.
prepend(object: T): OrderedSet(T)	The sequence consisting of *object*, followed by all elements in *self*.
insertAt(index : Integer, object : T) : OrderedSet(T)	The set consisting of *self* with *object* inserted at position *index*.
subOrderedSet(lower : Integer, upper : Integer) : OrderedSet(T)	The sub-set of *self* starting at number *lower*, up to and including element number *upper* (1<=lower<=upper<=size).
at(i : Integer) : T	The *i*-th element of *self* (1<=i<=size).
indexOf(object : T) : Integer	The index of *object* in the sequence.
first() : T	The first element in *self*.
last() : T	The last element in *self*.

Special Types

Taken from http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf

Type	Description
OclAny	Supertype for all types except for collection and tuple types. All classes in a UML model inherit all operations defined on OclAny.
OclVoid	The type OclVoid is a type that conforms to all other types. It has one single instance called *null*. Any property call applied on *null* results in *OclInvalid*, except for the operation oclIsUndefined(). A collection may have *null's*.
OclInvalid	The type OclInvalid is a type that conforms to all other types. It has one single instance called *invalid*. Any property call applied on *invalid* results in *invalid*, except for the operations oclIsUndefined() and oclIsInvalid().
OclMessage	Template type with one parameter T to be substituted by a concrete operation or signal type. Used in some postconditions that need to constrain the messages sent during the operation execution.
OclType	Meta type.

Operations Defined in OclAny

Taken from `http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf`

Operation	Description
=(object2 : OclAny) : Boolean	True if *self* is the same object as *object2*.
<>(object2 : OclAny) : Boolean	True if *self* is a different object from *object2*.
oclIsNew() : Boolean	Can only be used in a postcondition. True if *self* was created during the operation execution.
oclAsType(t : OclType) : OclType	Cast (type conversion) operation. Useful for downcast.
oclIsTypeOf(t: OclType) : Boolean	True if *self* is of type *t*.
oclIsKindOf(t : OclType) : Boolean	True if *self* is of type *t* or a subtype of *t*.
oclIsInState(s : OclState) : Boolean	True if *self* is in state *s*.
oclIsUndefined() : Boolean	True if *self* is equal to *null* or *invalid*.
oclIsInvalid() : Boolean	True if *self* is equal to *invalid*.
allInstances() : Set(T)	Static operation that returns all instances of a classifier.

Operations Defined in OclMessage

Taken from `http://www.di.uminho.pt/~jmf/MDSE/u2c.pdf`

Operation	Description
hasReturned() : Boolean	True if type of template parameter is an operation call, and the called operation has returned a value.
result()	Returns the result of the called operation, if type of template parameter is an operation call, and the called operation has returned a value.
isSignalSent() : Boolean	Returns true if the OclMessage represents the sending of a UML Signal.
isOperationCall() : Boolean	Returns true if the OclMessage represents the sending of a UML Operation call.
parameterName	The value of the message parameter.

Literature I

[UML09] UML Revision Task Force.
OMG Unified Modeling Language: Superstructure, February 2009.
http://www.omg.org/spec/UML/2.2/.
2

[UML10] UML Revision Task Force.
Object Constraint Language Specification, February 2010.
http://www.omg.org/spec/OCL/2.2/.
6

Literature II

[WK03] Jos Warmer and Anneke Kleppe.
The Object Constraint Language: Getting Your Models Ready for MDA.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
6