Design of Information Systems

OCL Collection Concepts and Collection Operations

Martin Gogolla
University of Bremen, Germany
Database Systems Group
Collections

- Collections common in modeling and programming languages
- "A collection (or container) is a **grouping** of some variable number of **data items** (possibly zero) that ... need to be operated upon together in some controlled fashion." Wikipedia
- Examples: set, list, multi-set (allowing duplicates), stack, ...
- UML collections: Set, Bag, Sequence, OrderedSet, Tuple
- Parametrized with element type(s) and access option (for Tuple)
Example collections in SocialNetwork

merkel.inviter: Set(Profile)

merkel.posting: Set(Posting)

merkel.posting.commenter: Bag(Profile)

-- !create merkel,putin,trump:Profile
Sequence{merkel,putin,trump}: Sequence(Profile)

OrderedSet{merkel,putin,trump}: OrderedSet(Profile)

Sequence{merkel,putin,trump,may}.yearE = Sequence{2005,2000,2016,2016}
-- yearE: year of first election; imaginable for example model


-- Paper::authors:OrderedSet(Author); more precise than
Sequence(Author)

Sequence{may,merkel}->collect(p|Tuple{L:p.lastN,I:p.initials}) =
  Sequence{Tuple{L='May', I='TM'},
        Tuple{L='Merkel',I='AM'}}:
Sequence(Tuple(L:String,I:String))
Example collections in ConferenceWorld
Collection parameters and collection syntax

- Type kinds with type parameters: \( \text{Set}(T), \text{Bag}(T), \text{Sequence}(T), \text{OrderedSet}(T), \text{Tuple}(A_1:T_1,\ldots,A_n:T_n); \) tuple component access \( A_i \)
- Abstract type kind (no instances): \( \text{Collection}(T) \), generalization of \( \text{Set}(T), \text{Bag}(T), \text{Sequence}(T), \text{OrderedSet}(T) \)
- Parameter actualization in order to build types
- Types (class model level) always written with parentheses \( ( ) \)

\[
\begin{align*}
\text{Set}(\text{Posting}), & \quad \text{Bag}(\text{Profile}), \\
\text{Sequence}(\text{Profile}), & \quad \text{OrderedSet}(\text{Integer}), \\
\text{Tuple}(L:\text{String},I:\text{String})
\end{align*}
\]

- Instantiations (object model level) always written with braces \( \{ \} \)

\[
\begin{align*}
\text{Set}\{\text{merkel, trump}\}, & \quad \text{Bag}\{\text{trump, putin, trump}\}, \\
\text{Sequence}\{\text{merkel, putin, trump}\}, & \quad \text{OrderedSet}\{2005, 2000, 2016\}, \\
\text{Tuple}\{L='Merkel', I='AM'\}
\end{align*}
\]

- Tuple access: \( \text{Tuple}\{L='Merkel', I='AM'\}.I='AM' \)
Collection properties (for homogeneous collections)

- Two criteria in order to distinguish between collections:
  (1) Insertion order relevance and (2) Insertion frequency relevance

- Is the insertion order relevant for distinguishing collections?
  \[
  \text{COL} -> \text{including}(E_1) -> \text{including}(E_2) = \text{COL} -> \text{including}(E_2) -> \text{including}(E_1)
  \]
  if required, collection is called order-blind, else order-aware

- Is the insertion frequency relevant for distinguishing collections?
  \[
  \text{COL} -> \text{includes}(E) \implies ( \text{COL} -> \text{including}(E) = \text{COL} )
  \]
  if required, collection is called frequency-blind, else frequency-aware

<table>
<thead>
<tr>
<th>frequency</th>
<th>order</th>
<th>blind</th>
<th>aware</th>
</tr>
</thead>
<tbody>
<tr>
<td>blind</td>
<td>Set(T)</td>
<td>OrderedSet(T)</td>
<td></td>
</tr>
<tr>
<td>aware</td>
<td>Bag(T)</td>
<td>Sequence(T)</td>
<td></td>
</tr>
</tbody>
</table>
Collection type hierarchy and properties

- order-blind and frequency-blind: Set(T)
- order-blind and frequency-aware: Bag(T)
- order-aware and frequency-aware: Sequence(T)
- order-aware and frequency-blind: OrderedSet(T)

- OCL 1.3 only had Set(T), Bag(T), Sequence(T)
- OCL 1.4 added OrderedSet(T)
- also used: order-insensible/-sensible, frequency-insensible/-sensible
Collection properties: Insertion order and frequency

Collection\{x,y\} = Empty{}->including(x)->including(y)

Set\{7,8\} = Set\{8,7\} = Set\{7,8,7\}

OrderedSet\{7,8\} <> OrderedSet\{8,7\} = OrderedSet\{7,8,7\}

Bag\{7,8\} <> Bag\{8,7\} = Bag\{7,8,7\}

Sequence\{7,8\} <> Sequence\{8,7\} = Sequence\{7,8,7\}

C->includes(E) implies C->including(E)=C

C->including(E1)->including(E2)=C->including(E2)->including(E1)
Collection properties

use> !C:=Set{Set{7,8}, Set{8,7},
          Set{7,8,8}, Set{8,7,7}}
use> ?C
Set{Set{7,8}} : Set(Set(Integer))

use> !D:=Set{Bag{7,8}, Bag{8,7},
          Bag{7,8,8}, Bag{8,7,7}}
use> ?D
Set{Bag{7,8}, Bag{7,7,8}, Bag{7,8,8}} : Set(Bag(Integer))

use> !E:=Set{OrderedSet{7,8}, OrderedSet{8,7},
          OrderedSet{7,8,8}, OrderedSet{8,7,7}}
use> ?E
Set{OrderedSet{7,8}, OrderedSet{8,7}} : Set(OrderedSet(Integer))

use> !F:=Set{Sequence{7,8}, Sequence{8,7},
          Sequence{7,8,8}, Sequence{8,7,7}}
use> ?F
Set{Sequence{7,8}, Sequence{8,7},
    Sequence{7,8,8}, Sequence{8,7,7}} : Set(Sequence(Integer))

use> ?Sequence{C->size(), D->size(), E->size(), F->size()}
Sequence{1, 3, 2, 4} : Sequence(Integer)
Collection operations on all collection kinds

Constructors and `destructors`
- Set{...}, Bag{...}, Sequence{...}, OrderedSet{...}
- including(...), excluding(...)

Basic boolean and integer query operations
- =, <>
- includes(...), excludes(...), includesAll(...), excludesAll(...)
- isEmpty(), notEmpty(), size(), count(...)

Advanced boolean query operations
- forAll(...), exists(...), one(...)
- isUnique(...)

Advanced collection-valued query operations
- select(...), reject(...)
- any(...)  
- union(...)  
- collect(...), collectNested(...)
- flatten()  
- sortBy(...)  

Complex query operations: iterate(...), closure(...)

Coercions: asSet(), asBag(), asSequence(), asOrderedSet()
Collection operations on special collection kinds

- `first()`, `last()`, `at(pos)`, `reverse()` for order-aware, i.e. `Sequence(T)`, `OrderedSet(T)`
- `subSequence(startPos,endPos)` on `Sequence(T)`
- `subOrderedSet(startPos,endPos)` on `OrderedSet(T)`
- `intersection(...)` for order-blind, i.e. `Set(T)`, `Bag(T)`
- `sum()`, `min()`, `max()` on `Collection(Integer)`, `Collection(Real)`
- Few further operations (e.g. `indexOf`): see OCL standard

Not mentioned yet (and to be discussed further down): collection operations in the context of generalization (e.g. for Chess example, `c:Character` and `c.oclIsTypeOf(Knight)`)
Demonstrating OCL expressions without having objects (Part A)

Constructors and `destructors'
- Set{7,8}, Bag{7,8,8}, Sequence{7,8,7}, OrderedSet{8,7,7}
- Set{}, Bag{}, Sequence{}, OrderedSet{}
- Set{7..9}, Bag{7..9}, Sequence{7..9}, OrderedSet{7..9}
- Set{}->including(8)->including(7), Bag{8,9,7,8,9}->excluding(9)

Basic boolean and integer query operations
- Set{7,8}=Set{8,7,8,7}, OrderedSet{7,8}<>OrderedSet{8,7,7}
  Set{7,8}<Bag{7,8}, OrderedSet{7,8}<Sequence{8,7,7}
- Set{7,8}->includes(8), Set{7,8}->excludes(9),
  Set{7,8}->includesAll(Set{8,8,7,7}), Set{7,8}->excludesAll(Set{6,9})
- Set{}->isEmpty(), Set{7,8}->notEmpty(), Set{8,8,7,7}->size()=2
  Set{7,8,7}->count(7), Bag{7,8,7}->count(7)
  Sequence{7,8,7}->count(7), OrderedSet{7,8,7}->count(7)
Demonstrating OCL expressions without having objects (Part B)

Advanced boolean query operations
- Set{7..9}→\texttt{forAll}(i|i\geq0), Bag{7..9}→\texttt{exists}(i|i\mod(2)=0)
- Sequence{7..9}→\texttt{one}(i|i\mod(2)=0)
- OrderedSet{-9..-8}→\texttt{including}(8)→\texttt{including}(9)→\texttt{isUnique}(i|i*i)=false

Advanced collection-valued query operations
- Set{21..42}→\texttt{select}(i|i\mod(3)=0 \text{ and } i\mod(7)=0)
- Bag{21..42}→\texttt{reject}(i|i\mod(2)=0 \text{ or } i\mod(3)=0)
- Set{21..42}→\texttt{any}(i|i\mod(2)=1)
- Set{7,8,8}→\texttt{union}(Set{9,9,8}), Bag{7,8,8}→\texttt{union}(Bag{9,9,8})
  Sequence{7,8,8}→\texttt{union}(Sequence{9,9,8})
  OrderedSet{7,8,8}→\texttt{union}(OrderedSet{9,9,8})
- Set{-2..2}→\texttt{collect}(i|i*i), Set{-2..2}→\texttt{collect}(i|Sequence\{i,i*i\})
  Set{-2..2}→\texttt{collectNested}(i|Sequence\{i,i*i\})
- Set{-2..2}→\texttt{collectNested}(i|Sequence\{i,i*i\})→\texttt{flatten}()
- Set{-6,-5,-4,7,8,9}→\texttt{sortedBy}(i|i*i)
Complex query operations

- Set{-2..2} \rightarrow \textbf{iterate}(i: \text{Integer}; r: \text{Set(Sequence(OclAny))}=\text{Set}{}) | 
  r \rightarrow \text{including}(\text{Sequence}\{i, i*i, \text{if } i \text{.mod}(2)=0 \text{ then 'E' else 'O' endif}))

- Capitals: M[adrid], P[aris], A[msterdam], B[erlin], Z[urich], V[ienna]
  let TupleSet=
    Set\{Tuple\{s: 'M', t: 'P'\}, Tuple\{s: 'P', t: 'A'\}, Tuple\{s: 'A', t: 'B'\},
    Tuple\{s: 'M', t: 'Z'\}, Tuple\{s: 'Z', t: 'V'\}, Tuple\{s: 'V', t: 'B'\}\} in
  TupleSet\rightarrow \textbf{closure}(T1|
    TupleSet\rightarrow \text{select}(T2|T1.t=T2.s)\rightarrow
    \text{collect}(T2|\text{Tuple\{s:T1.s, t:T2.t\}}))

  \text{select} =
  +-----------------------+
  |                     |
  Tuple\{T1.s, T1.t\}   Tuple\{T2.s, T2.t\}
  |                     |
  +-----------------------+

  \text{collect constructs new, transitive tuple}
Demonstrating OCL expressions without having objects (Part D)

Coercions
- Sequence{8,7,8} -> asSet() = Set{8,7}
- OrderedSet{8,7,8} -> asBag() = Bag{8,7}
- Set{7,8} -> asSequence() = Sequence{8,7}
  or Set{7,8} -> asSequence() = Sequence{7,8}
- Bag{8,8,7,7} -> asOrderedSet() = OrderedSet{7,8}
  or Bag{8,8,7,7} -> asOrderedSet() = OrderedSet{8,7}
- Set{-2..2} -> collect(i | i*i) -> asSet()
Collection operation iterate for iterations

- COLEXPR->iterate(ELEMVAR:ELEMTYPE; RESVAR:RESTYPE=INITEXPR | ITEREXPR)

- COLEXPR, INITEXPR, ITEREXPR: OCL expression
  ELEMVAR, RESVAR: OCL variables
  ELEMTYPE, RESTYPE: OCL types
  ITEREXPR may use ELEMVAR, RESVAR; ITEREXPR not forced to do so

  type (COLEXPR) in
  \{Set(ELEMTYPE), Bag(ELEMTYPE), Sequence(ELEMTYPE), OrderedSet(ELEMTYPE)\}
  type (INITEXPR) = type (ITEREXPR) = RESTYPE

- Also allowed: COLEXPR->iterate(ELEMVAR; RESVAR:RESTYPE=INITEXPR | ITEREXPR)
  i.e., ':ELEMTYPE' is optional

- Collection operations can be expressed with iterate

- Example

  ibm.worker->exists(p:Person | p.fName='Bob')

  ibm.worker->iterate(p:Person; bobEx:Boolean=false | bobEx or p.fName='Bob')

  COLEXPR ibm.worker
  ELEMVAR p
  ELEMTYPE Person
  RESVAR bobEx
  RESTYPE Boolean
  INITEXPR false ibm.worker = Set{ada,bob} ->
  ITEREXPR bobEx or p.fName='Bob' false or ada.fName='Bob' or bob.fName='Bob'
- iterate Evaluation in Java-like Pseudo Code

\[ \text{COLEXPR} \rightarrow \text{iterate}(\text{ELEMVAR} : \text{ELEMTYPE}; \ \text{RESVAR} : \text{RESTYPE} = \text{INITEXPR} \mid \text{ITEREXPR}) \]

```java
RESTYPE iterate() {
    ELEMTYPE ELEMVAR;
    RESTYPE RESVAR = INITEXPR;
    for (Iterator i = COLEXPR.iterator(); i.hasNext();)
    {
        ELEMVAR = (ELEMTYPE)i.next();
        RESVAR = ITEREXPR;
    }
    return RESVAR;
}
```

- Expressing other collection operation with iterate; given COL:Set(T)

\[ \text{COL} \rightarrow \text{select}(e \mid p(e)) \Rightarrow \]
\[ \text{COL} \rightarrow \text{iterate}(e; \ r: \text{Set}(T)=\text{Set}{} \mid \text{if } p(e) \text{ then } r \rightarrow \text{including}(e) \text{ else } r \text{ endif}) \]

\[ \text{COL} \rightarrow \text{collect}(e \mid t(e)) \Rightarrow \text{COL} \rightarrow \text{iterate}(\ldots) \]

\[ \text{COL} \rightarrow \text{forAll}(e \mid p(e)) \Rightarrow \text{COL} \rightarrow \text{iterate}(e; \ r:\text{Boolean}=\text{true} \mid \text{r and p(e)}) \]

\[ \text{COL} \rightarrow \text{iterate}(e; \ r:\text{Boolean}=\text{true} \mid \text{false}) \Leftrightarrow \text{COL} \rightarrow \text{ColOpXYZ}() \]

\[ \text{COL} \rightarrow \text{size()} \Rightarrow \text{COL} \rightarrow \text{iterate}(e; \ sz:\text{Integer}=0 \mid sz+1) \]

...
Transitive closure

In mathematics, the **transitive closure** of a binary relation $R$ on a set $X$ is the smallest relation on $X$ that contains $R$ and is transitive.

For example, if $X$ is a set of airports and $xRy$ means "there is a direct flight from airport $x$ to airport $y"$ (for $x$ and $y$ in $X$), then the transitive closure of $R$ on $X$ is the relation $R^+$ such that $x R^+ y$ means "it is possible to fly from $x$ to $y$ in one or more flights". Informally, the transitive closure gives you the set of all places you can get to from any starting place.

**Existence and description**

For any relation $R$, the transitive closure of $R$ always exists. To see this, note that the intersection of any family of transitive relations is again transitive. Furthermore, there exists at least one transitive relation containing $R$, namely the trivial one: $X \times X$. The transitive closure of $R$ is then given by the intersection of all transitive relations containing $R$.

For finite sets, we can construct the transitive closure step by step, starting from $R$ and adding transitive edges. This gives the intuition for a general construction. For any set $X$, we can prove that transitive closure is given by the following expression

$$R^+ = \bigcup_{i=1}^{\infty} R^i,$$

where $R^i$ is the $i$-th power of $R$, defined inductively by

$R^1 = R$

and, for $i > 0$,

$$R^{i+1} = R \circ R^i$$

where $\circ$ denotes composition of relations.
Collection operation closure for transitive closure and cycles

- COL : Collection(C); C::CLOSURE_TERM:Collection(C)
  CLOSURE_TERM: role, attr, query operation or collection operation on them
- COL->closure( CLOSURE_TERM )
  COL->closure( ELEMVAR | CLOSURE_TERM )
  COL->closure( ELEMVAR:ELEMTYPE | CLOSURE_TERM )
- Given C::term:Set(C) and c:C :
  c.term->closure(term) = transitive closure; c included if reachable by term
  Set{c}->closure(term) = reflexive, transitive closure; c always included
Collection operation closure – Further examples

```bash
use> ?ada.friends()                  use> ?ada.friends()->closure(friends())
Set{bob,cyd} : Set(Profile)          Set{ada,bob,cyd} : Set(Profile)
use> ?bob.friends()                  use> ?bob.friends()->closure(friends())
Set{ada} : Set(Profile)              Set{ada,bob,cyd} : Set(Profile)
use> ?cyd.friends()                  use> ?cyd.friends()->closure(friends())
Set{ada} : Set(Profile)              Set{ada,bob,cyd} : Set(Profile)
```

```bash
use> ?ada.inviter->union(ada.invitee)
Set{bob,cyd} : Set(Profile)
use> ?bob.inviter->union(bob.invitee)
Set{ada} : Set(Profile)
use> ?cyd.inviter->union(cyd.invitee)
Set{ada} : Set(Profile)
```

```bash
use> ?ada.inviter->union(ada.invitee)->closure(inviter->union(invitee))
Set{ada,bob,cyd} : Set(Profile)
use> ?bob.inviter->union(bob.invitee)->closure(inviter->union(invitee))
Set{ada,bob,cyd} : Set(Profile)
use> ?cyd.inviter->union(cyd.invitee)->closure(inviter->union(invitee))
Set{ada,bob,cyd} : Set(Profile)
```

```bash
use> ?dan.inviter
Set{gil} : Set(Profile)
use> ?dan.inviter->closure(inviter)
Set{dan,eve,flo,gil} : Set(Profile)
```
Collection operation closure – Classical example: Acyclic parenthood

USE brings original, short expression ...
... closure(parent)
into a form with an explicit variable like ...
... closure(p [:Person] | p.parent)

class Parent:
    0..2 parent
    * child

context p:Person inv acyclicParenthood:
p.parent->closure($elem0:Person | $elem0.parent)->excludes(p)

LEFT OBJECT DIAGRAM
bob.child->closure(child)
Set(dan,eve) : Set(Person)

bob.parent->closure(parent)
Set(ada) : Set(Person)

RIGHT OBJECT DIAGRAM
bob.child->closure(child)
Set(ada,bob,cyd,dan,eve) : Set(Person)

bob.parent->closure(parent)
Set(ada,bob,eve) : Set(Person)

bob:Person
  parent
cyd:Person
  parent
dan:Person
  parent
eve:Person
  parent

Class invariants

Invariant | Satisfied
---|---
Person::acyclicParenthood | true
Person::acyclicParenthood | false
Collection operation closure – Analysis with USE Evaluation Browser

- Double-clicking the failing invariant opens the Evaluation Browser Window

- Window can be tuned through context menu and bottom selection box to explore which objects contribute to invariant failure
Thanks for your attention!