Constraint Specification

Consider the UML diagrams as we've seen them so far:

- Could an programmer be tasked with writing the code based *only* on the UML?
- Could we use it for design verification?
- Could we use it to define test cases and expected results?
- Could we automatically generate the code from it?

These are all things we should be able to do with a complete model.

Constraints (cont'd)

Although the UML diagrams are precise, they're not complete — we need to add more information.

- Class diagrams should define
  - Initial values;
  - Restrictions on values, associations, multiplicities;
  - Relationships between values, either static or dynamic;
  - Meaning of operations;
  - Derived quantities.
- Interaction diagrams need
  - Conditions,
  - Actual parameter values.
- State diagrams need
  - Guards,
  - Restrictions on states,
  - Targets of actions,
  - Change events,
  - Actual parameter values.
- Use cases need pre and post conditions.

Object Constraint Language

- Part of UML.
- Both constraint and query language.
  - A *constraint* is a restriction on one or more values of (part of) an object-oriented model or system.
  - A *query* is an expression that defines a value or collection of values in a model. (A Boolean query can be used as a constraint.)
- Mathematically based but uses few special symbols.
- Strongly typed — all expressions have a type. It is an error to use expressions of the wrong type (e.g., use int where Boolean is expected).
- Declarative — specifies *what* rather than *how*.

Context

- All OCL expressions are with respect to a *context* — the model entity for which the expression is defined.
- Usually the context is a class, interface, datatype or component.
- Could also be an operation, attribute or instance.
- The type of the context is called the *contextual type*.
- OCL expressions are evaluated with respect to an instance — the *contextual instance*.
- *self* is an OCL expression (keyword) whose value is the contextual instance.

OCL Expressions

Initial values — expressions that give initial value to attributes.

```
context Credit::grade
init: 0
```

Derivation rules — expression that defines how the value of a derived attribute or association is computed.

```
context Student::average
derive: Credit.grade->sum() / Credit->size()
```

Query Operations — expressions that define the value for operations that don't change state of system.

```
context Student::enrolledCourses(yr : integer,
sem : integer) : Set(Course)
body: self.enrolledIn->select(year = yr and
semester = sem )->collect(Course)
```

OCL Expressions (cont'd)

Attribute Definitions — Expression that defines an attribute in addition to those in the diagrams.

```
context Credit
def: points : integer =
 if grade >= 80 then 4
 else if grade < 80 and grade >= 65 then 3
 else if grade < 65 and grade >= 55 then 2
 else if grade < 55 and grade >= 50 then 1
 else 0
 endif
```

OCL Expressions (cont'd)

Invariants — Constraint that must be true

- upon completion of the constructor and
- upon completion of every public operation

```
context Section
inv qualifiedInstructor:
instructor.canTeach->includes(Course)
```

Pre-condition — Constraint that must be true immediately before an operation starts its execution.

Post-condition — Constrint that must be true at the moment when an operation ends its execution. (Note: use @pre to denote value at start.)

```
context Student::enroll(sec :  Section) :  void
pre:  Credit.Section.Course->includesAll(
                              sec.Course.prerequisite)
post:  enrolledIn->includes(sec)
```

OCL Elements

Every OCL expression has a type.

Basic types: Integer, Real, String and Boolean.

Collection types: Set, Bag, OrderedSet, Sequence.

User defined types (in model):

- *Query* operations — those that don't change the state of any object.
- Instance attributes: `instance.attribute`
- Class attributes/operations: `Class::attribute`
- Associations and aggregations — use role name or type name as attribute. (Multiplicity greater than one gives collection.)

Boolean Operators

```
a or b
a and b
a xor b
not a
a = b
a <> b
a implies b
if bool expr then expr else expr endif
```

Integer and Real operators

```
a = b
a <> b
a < b
a > b
a <= b
a >= b
a + b
a - b
a * b
a / b
```

```
a.mod(b)
a.div(b)
a.abs()
a.max(b)
a.min(b)
a.round()
a.floor()
```

String operators

```
s1.concat(s2)
s1.size()
s1.toLower()
s1.toUpper()
s1.substring(s, f)
s1 = s2
s1 <> s2
```

Literals written with enclosing single quotes.

Collections

Set	Unordered, no duplicates.
OrderedSet	Ordered, no duplicates.
Bag	Unordered, may contain duplicates.
Sequence	Ordered, may contain duplicates.

Collection Constants

Define explicit instances:

```
Set { 1, 2, 3, 99 }
OrderedSet { 'John', 'Mary', 'Jane' }
Sequence { 'ape', 'nut' }
Bag { 1, 3, 4, 3 5 }
```

Collection Types A type that is "collection of type":

```
Set(Student)
```

Collections (cont'd)

Collection Operations Standard operations defined on collections.

Denoted by *collection->operation*

Basic operations

`c->count(o)`	Number of occurances of o in c
`c->excludes(o)`	True iff o is not an element of c
`c->excludesAll(c2)`	True iff all of c2 are not in c
`c->includes(o)`	True iff o is an element of c
`c->includesAll(c2)`	True iff all of c2 are in c
`c->isEmpty()`	True if c contains no elements.
`c->notEmpty()`	True if c contains one or more elements.
`c->size()`	number of elements in c
`c->sum()`	Addition of all elements in c

Operations with variant meaning

`c1 = c2`	`c1` and `c2` contain same elements (*in the same order for ordered collections*)
`c1 <> c2`	Not equals.
`c1 - c2`	Remove elements in `c2` from `c1` if present (Set and OrderedSet only)
`c->flatten()`	Merge collection of collection into 'flat' collection (default behaviour). For ordered collections of unordered collections (e.g., Sequence of Sets) the resulting order is non-deterministic.
`c->excluding(o)`	Remove all occurances of `o` from `c`.
`c->including(o)`	Add `o` to `c`. (No change for Set or OrderedSet already containing `o`).

Operations with variant meaning (cont'd)

`c1->union(c2)`	Merge collections. Ordered collections can only be unioned with ordered collections (append c2 to c1).
`c1->intersection(c2)`	Only elements in both c1 and c2. Not valid for ordered collections.
`c1->symmetricDifference(c2)`	sets only. Gives collection of elements in exactly one of c1 or c2.

Operations with variant meaning (cont'd)

`c->asBag()`	Convert to bag (order is lost)
`c->asOrderedSet()`	Convert to ordered set (duplicates lost, random order if c is unordered)
`c->asSequence()`	Convert to sequence (random order if c is unordered).
`c->asSet()`	Convert to set (order and duplicates lost)

Operations on ordered collections

`c->append(o)`	Append to end.
`c->prepend(o)`	Insert at begining.
`c->at(i)`	ith element.
`c->first()`	first element.
`c->last()`	last element.
`c->indexOf(o)`	Index of first occurance of o (indexed from 1)
`c->insertAt(i, o)`	Insert o at index `i`.
`c->subOrderedSet(l, u)`	OrderedSet only.
`c->subSequence(l, u)`	Sequence only.

Iterators

Evaluate expresssion on elements of collection.

Iterator variable can be declared in `exp`:

`Course.prerequisites->collect( c | c.number )`

`c->exists(exp)`	True iff there is at least one element in `c` for which `exp` is true.
`c->forAll(exp)`	True iff `exp` is true for every element in `c`.
`c->isUnique(exp)`	True iff `exp` has a unique value for every element in `c`.
`c->one(exp)`	True iff there is exactly one element in `c` for which `exp` is true.
`c->any(exp)`	A random element for which `exp` is true.

Iterator

```
collection->iterate( element : Type1;
                     result : Type2 = <expr1> |
                     <expr2> )
```

- `element` is iterator variable
- `Type1` is type of elements in `collection`
- `result` is *accumulator*
- `<expr1>` is initial value for `result`
- `<expr2>` is an expression including `element` and `result`.
- Semantics: for each `element` in `collection`, `<expr2>` is evaluated using 'previous' value of `result`.

```
Set(1, 2, 3)->iterate(i:Integer, sum:Integer = 0 |
sum+i)
```

Collection Constructors

`c->collect(exp)`	All objects resulting from `exp` on elements of `c`.
`c->collectNested(exp)`	Colletion of collectiosn resulting from `exp` on elements of `c`.
`c->reject(exp)`	Subcollection of `c` containing elements for which `exp` is false.
`c->select(exp)`	Subcollection of `c` containing elements for which `exp` is true.
`c->sortedBy(exp)`	Ordered Subcollection of c with elements ordered according to increasing `exp`.

Local varialbles

Help to make expressions easier to read:

```
let <var> :  <Type> = <defn>
    in <expr>
```

- `<var>` is a variable name.
- `<Type>` is a type.
- `<defn>` is an expression of type `<Type>`.
- `<expr>` is an expression involving `<var>`.
- More than one var can be defined.

Tuples

A collection of named parts (think `struct` in C):

```
Tuple { <name1> : <Type1> = <val1>,
        <name2> : <Type2> = <val2>, ...}
TupleType(<name1> : <Type1>, <name2> : <Type2>,
...)
```

Type casting

`o.oclAsType(Type2)` — evaluates `o` with type `Type2`.

Only applicable when `Type2` is a subtype of the type of `o`.

Postconditions

In postconditions we need to express what has been changed by the operation — we need to compare two states (before and after operation execution).

Postcondition constructs:

`a@pre`	The value of a at the start of execution of the operation.
`result`	The value returned by the operation.
`v->oclIsNew()`	True iff `v` is constructed during execution of the operation.
`a^op(arg)`	*isSent*: True iff the operation has sent (called) `op(arg)` on a during its execution. Argument value may be unspecified — denoted by "?".
`a^^op(arg)`	*message operator*: The sequence of messages sent that match `op(arg)` during the execution of the operation. Type is `Sequence(OclMessage)`.

OclMessage Operations

- Special type `OclMessage`
- Wraps any operation call or signal transmission.
- Signal is asynchronous (no return value).
- Operation can be synchronous or asynchronous.

`m.hasReturned()`	True iff `m` has finished executing.
`m.result()`	Return value of `m`.
`m.isSignalSent()`	True iff `m` is a signal.
`m.isOperationCall()`	True iff `m` is an operation call.

OclAny Type

`OclAny` is a supertype for all types.

Operations on `OclAny` (inherited by all subtypes)

`o.oclIsUndefined()`	True iff `o` is undefined.
`o.oclIsTypeOf(<Type>)`	True iff `o` of type `<Type>`.
`o.oclIsKindOf(<Type>)`	True iff `o.oclIsTypeOf(<Type>)` or `o` is an instance of a subtype of `<Type>`.
`o.oclInState(<sname>)`	True iff `o` is in the state named `<sname>`. `o` must have associated state chart.
`type::allInstances()`	The set of all instances of type. (usage discouraged)

References

[1] Jos Warmer and Anneke Kleppe.
Object Constraint Language: Getting Your Models Ready for MDA.
Addison-Wesley, second edition, 2003.