Topics Outline

- Motivation
- Solution
  - Metamodel
  - Code Generation
  - Type Checker
  - Constraint Checker
- Validation
  - Order System Example
- Future Work
- Conclusion
Motivation

- MOF/UML:
  - Use OCL and natural language to constrain the metamodel

- Are OCL and natural language constraints included in the metamodel itself?

- True bootstrapping would assume so…
Motivation:  
Current Situation in MOF 2.0

- The constraint specification is a ValueSpecification
  - A ValueSpecification identifies values in a model
  - Can be an Expression (e.g. $a + b = 3$)
  - Can be an OpaqueExpression (e.g. an OCL statement)

- Where does it go from there?
Motivation:

Goals

- Define a metamodel for Class Diagrams in which constraints are also metamodeled
- Be able to check a model instance against a model defined in this “new” formalism
- Start using pyGK
Solution: General Approach (1)

ClassDiagramWithConstraints

- Class Diagram
- Constraint Language

references

Re-metamodel Class Diagrams
- Including a constraint language
  - Abstract (and concrete) syntax
- Inspired by MOF and OCL

AToM³

produces

Model + Constraints (ASG)

ClassDiagramWithConstraints

Modeling Environment

+ Constraints (ASG)
Solution:
General Approach (2)

Model + Constraints (ASG) → ASG2pyGK → Model + Constraints (pyGK)

model_pyGK.py

```python
def model_MDL():

    def constraint1(context):
        ...
```

model_chkr.py

```python
def model_MDL_chkr():

    checks type

    checks constraints
```

Solution:
Class Diagrams With Constraints (1)
Solution:
ClassDiagramsWithConstraints (2)
Solution: Overview of pyGK (1)

- The **Python** **Graph** **Kernel**
- Developed by Marc Provost
- An easy to use API for graph representation
Solution:
Overview of pyGK (2)

- _id: unique identifier
- _label: “type”

PrimitiveTypes:
- Int, Float, Bool, String, List
- SymbolTable / AttrNode

Straightforward functions to:
- Construct a graph:
  - add(), connect()
- Traverse a graph:
  - BFS, DFS

Solution:
Converting ASG model to pyGK

E.g.

Class Diagram in ASG format

```
model = Graph(ID="model", label="ClassDiagram")
A = AttrNode(ID="A", label="Class")
A["att1"] = Int()
B = AttrNode(ID="B", label="Class")
B["att2"] = String()
C = AttrNode(ID="C", label="Class")
C["att3"] = Float()
hasB = AttrNode(ID="hasB", label="Association")
hasB["srcMultMin"] = Int(value=1)
hasB["srcMultMin"] = Int(value=1)
hasB["trgMultMin"] = Int(value=0)
hasB["trgMultMin"] = String(value="N")
CInheritsB = AttrNode(ID="CInheritsB", label="Inherit")
```

// Add and connect nodes in model

Class Diagram in pyGK format

Iterate through the ASGNodes and generate…
Solution: Converting ASG constraint to pyGK

E.g

Constraint expression in ASG format

```
Person
\+\text{Age: Int}
```

Constraint expression in pyGK format

```python
constraints = model(ID="constraints", label="Constraints")
Invariant1 = AttrNode(ID="Invariant1", label="Invariant")
comp1 = AttrNode(ID = "comp1", label = "ComparisonOp")
PersonAge = AttrNode(ID="PersonAge", label="AttributeCall")
PersonAge["calledAttributeType"] = Int()
PersonAge["calledAttributeName"] = String(value="Age")
PersonAge["owningClassifier"] = String(value="Person")
comp1["operator"] = String(value = "GREATER_{EQ}")
comp1["argument1"] = PersonAge
comp1["argument2"] = Int(value=0)
Invariant1["bodyExp"] = comp1
Invariant1["context"] = String(value="Person")
```

// Add and connect nodes in constraints
Solution: Saving to file (1)

- Define an API for constructing Python code:
  - A sort of an AST interface for Python
  - E.g. an Assignment Statement is
    
    AssignmentStmt ::= LHS \( \text{“} = \text{“} \) RHS
    LHS ::= LiteralStmt
    RHS ::= LiteralStmt | OperationStmt

- Using this API, we can generate code constructs like *If* statements and function calls

- Also supports indentation for writing out Python code
Solution: Saving to file (2)

```python
lit1 = LiteralStmt("a")
lit2 = LiteralStmt("in")
lit3 = LiteralStmt("c")
listLit = LiteralStmt("[0, 1, 2, 3]")
trueLit = LiteralStmt("True")
zeroLit = LiteralStmt("0")
imp1 = ImportStmt(LiteralStmt("pack"), LiteralStmt("mod"))
def1 = DefStmt(LiteralStmt("foo"), [lit2])
blk1 = BlockStmt([imp1])
blk1.appendReturnCarriage()
blk1.appendStmt(def1)
ass1 = AssignmentStmt(lit1, lit2)
plus1 = NaryOpStmt(LiteralStmt("+") , [lit1, lit2])
ass2 = AssignmentStmt(LHS=lit3, RHS=plus1)
eq1 = BinaryOpStmt(LiteralStmt("=="), [lit3, trueLit])
call1 = FunctionCallStmt(context=None, fnName=LiteralStmt("len"), arguments=[listLit])
less1 = BinaryOpStmt(LiteralStmt("<"), [call1, zeroLit])
if1 = IfStmt(eq1, BlockStmt([ReturnStmt(trueLit)]), BlockStmt([ReturnStmt(less1)]))
blk2 = BlockStmt([ass1, ass2, if1])

print blk1.toString(0)
print blk2.toString(1)
```

Output:
```
from pack import mod

def foo(in):
a = in
c = (a + in)
if (c == True):
    return True
else:
    return (len([0, 1, 2, 3]) < 0)
```
Solution: Saving to file (3)

- Saving the model:
  - Define a function:
    
    ```python
def <modelName>_MDL():
```
  - Spit out the pyGK statements that can rebuild the model

- Saving the constraints:
  - For each constraint, define a function:
    
    ```python
def <constraintName>(context):
```
  - Turn the pyGK format into Python code using the syntax API
Solution: Type Checking

- Checks that a model instance conforms to a model
- It entails checking
  - That every instance element corresponds to a meta-element
  - That every instance element owns only properties that its meta-element can own
  - That every association in the model is respected
...every instance element corresponds to a meta-element

- For every (pyGK) Node in the instance,
  - Check that the model contains a Node whose id corresponds to the instance Node’s label

- E.g.

**In model:**

<table>
<thead>
<tr>
<th>A</th>
<th>+att1: Int</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**In instance:**

<table>
<thead>
<tr>
<th>myA : A</th>
<th>+att1 = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

AttrNode(id="A", label="Class")

AttrNode(id="myA", label="A")
... every instance element owns only properties that its meta-element can own

For each key in an AttrNode of the instance,
- Check that the corresponding meta-element, or a super type of the meta-element, has the same key
- Check that the values for the corresponding keys have the same type

E.g.

**In model:**

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>+att1: Int</td>
</tr>
</tbody>
</table>

A = AttrNode(id="A", label="Class")
A("att1") = Int()

**In instance:**

<table>
<thead>
<tr>
<th>myA : A</th>
</tr>
</thead>
<tbody>
<tr>
<td>+att1 = 0</td>
</tr>
</tbody>
</table>

myA = AttrNode(id="myA", label="A")
myA("att1") = Int(value=0)
...every association in the model is respected (1)

- Check that their instances attach the correct types, as permitted by the model
  - For every association,
    - Get the hierarchy of the source and built a list of IDs
    - Do the same for the target
    - Check that every instance connects
      - a source whose label is in the source ID list
      - a target whose label is in the target ID list

- Complexity: Inheritance
...every association in the model is respected (2)

- Check that their instances respect the multiplicities
  - Build a tuple table of all the instance links
  - Check:
    - $\text{srcMultMin} \leq \# \text{source instances} \leq \text{srcMultMax}$
    - $\text{trgMultMin} \leq \# \text{target instances} \leq \text{trgMultMax}$
Solution: Constraint Checking

- In type checking, the checker can be written offline
  - Applicable to any model

- For constraint checking, the checker is specific to the model
  - So, we need to generate something that will check each constraint against every instance of the constraint’s context
  - Call the generated Python function
  - Don’t forget sub types!
Validation

○ So does all this work???

○ Let us see the solution in action

○ Order System Example
Validation:
What do we have now?

- A constraint language that is included within the Class Diagram metamodel
- "Static" checking of model instances
  - A client application of the checker would input a model instance that is considered to be "stable" at that point
- A concrete application for pyGK
Validation:
Limitations & Future Work

- What about operations?
- N-ary associations
- Visually surcharged models
  - But the abstract syntax tree will look something like that…
- Expressiveness of the constraint language
  - Add more constructs like \textit{for} loop or \textit{select} operation
- Tests on bigger models
  - Performance issues?
More Future Work

- Check multiplicities as constraints
  - So Type Checker would be a pure structural check

- Generation of modeling environment
  - Generate a modeling environment from `ClassDiagramWithConstraints` but without the constraint language

- Bootstrapping
  - Re-metamodel `ClassDiagramWithConstraints` in itself
    - Defining the constraints in the constraint language itself
Acknowledgements

Special thanks to:

- Dr. Hans Vangheluwe
- Dr. Juan de Lara for ClassDiagram formalism in AToM³
- Marc Provost for help on pyGK and metamodeling
References


