

UML object constraint language in Meta-Modeling

WeiBin Liang
School of Computer Science
McGill University

September 5, 2002

Contents

1	Project overview	3
1.1	Specifies constraints in ATOM3	3
1.1.1	An rough introduction to ATOM3	3
1.1.2	How are constraints currently specified in ATOM3	4
1.1.3	Why is using OCL a better way to specified constraints	4
1.2	Project requirements	4
1.3	Two possible approaches to the goal	4
1.3.1	A traditional way	5
1.3.2	A new way with graph grammar	6
1.4	Why did we choose the second way	6
2	The works I have done for this project	7
2.1	An OCL presentation	7
2.2	Prototypes of meta-model of OCL expression with ER-model	7
2.2.1	Prototype 1	7
2.2.2	Prototype 2	8
2.3	Prototypes of an OCL parser	12
2.3.1	Prototype 1	12
2.3.2	Prototype 2	14
2.3.3	Prototype 3	14
3	Future works	17
4	Acknowledgements	17

List of Figures

1	OCL2Python process	5
2	A traditional way to do type check and code generation	5
3	Type check and code generation in ATOM3	6
4	Class diagram of Prototype 1	8
5	A meta-model of Prototype 1	9
6	An abstract syntax tree for the example in Prototype 1	9
7	Class diagram of Prototype 2	10
8	A meta-model of Prototype 2	10
9	An abstract syntax tree for the example in Prototype 2	11
10	Dependency chart of prototype1 of the OCL parser	13
11	The output abstract syntax tree	15

1 Project overview

I got a NSERC award to do this project on UML object constraint language in meta-modeling under the supervision of Prof. Hans Vangheluwe in summer, 2002. Here below is a brief description of the project.

1.1 Specifies constraints in ATOM3

The formalisms used in ATOM3, such as ER diagram, DEVS, PetriNet, are graphical languages. So they are not sufficient to specify complicated constraints for the model. Textual constraints needs to be added in.

1.1.1 An rough introduction to ATOM3

AToM3 is a tool for multi-paradigm modelling under development at the Modelling, Simulation and Design Lab (MSDL) in the School of Computer Science of McGill University. It is developed in close collaboration with Prof. Juan de Lara of the School of Computer Science, Universidad Autnoma de Madrid (UAM), Spain. AToM3 stands for *A Tool for Multi-formalism and Meta-Modelling*.

The two main tasks of AToM3 are meta-modelling and model-transforming. Meta-modelling refers to the description, or modelling of different kinds of formalisms used to model systems (although we have focused on formalisms for simulation of dynamical systems, AToM3's capabilities are not restricted to these.) Model-transforming refers to the (automatic) process of converting, translating or modifying a model of a given formalism, into another model that might or might not be in the same formalism.

In AToM3, formalisms and models are described as graphs. From a meta-specification (in the ER formalism) of a formalism, AToM3 generates a tool to manipulate (create and edit) models described in the specified formalism. Model transformations are performed by graph rewriting. The transformations themselves can thus be declaratively expressed as graph-grammar models.

Some of the meta-models currently available are: Entity-Relationship, GPSS, Deterministic Finite state Automata, Non-Deterministic Finite state Automata, Petri Nets, Data Flow Diagrams and Structure Charts. Typical model transformations include model simplification (e.g., state reduction in Finite State Automata), code generation, generation of executable simulators based on the operational semantics of formalisms, as well as behaviour-preserving transformations between models in different formalisms.

1.1.2 How are constraints currently specified in ATOM3

At the time being, users of ATOM3, the model designers, need to write Python code fragments to specified the constraints such that these code fragments will be run and tested as pre-conditions during the building of the model and be tested as post-conditions when the model is saved.

But there are several drawbacks to specified constraints in this way. First of all, it requires the user be comfortable with programming in Python. This will prevent a lot of potential user from actually using ATOM3. Secondly, the user must know how the model info is stored in ATOM3 and directly manipulates the internal data structures. And this is really bad, because it violate the rule of object-oriented design, information and implementation hiding. If the designer of ATOM3 later find a better format in which the model info to be stored, it adds extra burdens to either the user or the designer, since there is no easy way to make the former Python codes work in the new ATOM3 environment. Third, this will distract the users from focusing on the model (or meta-model) they are designing.

1.1.3 Why is using OCL a better way to specified constraints

We do have the following benefits using OCL to specified constraints:

1. It hides the underlying data structures of ATOM3 environment from the user.
2. It frees the user from a lot of detailed programming works. As a matter of fact, the average business and system modeller may not be familiar with data structures and programming techniques.
3. OCL is formal language that remains easy to read and write.
4. OCL is a pure expression language which guarantee no side effects.

1.2 Project requirements

We will study the syntax and semantic of UML object constraint language in a formal way. We will also define a mapping between the most declarative OCL constructs, like `forAll`, and the Python codes which carry out the actual checking. Then, we will design and implement an OCL to Python translator. Finally, we will integrated this translator into ATOM3 meta-modeling environment.

1.3 Two possible approaches to the goal

A process of translating an OCL constraints to the equivalent Python codes that will carry out the actual checking is illustrated in figure 1.

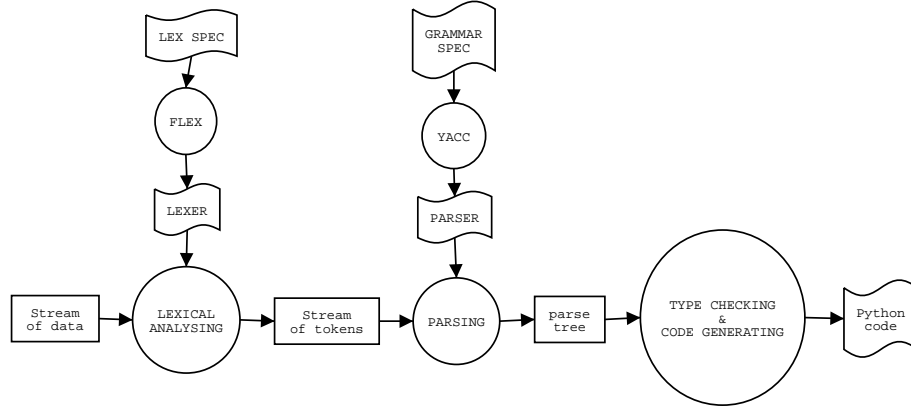


Figure 1: OCL2Python process

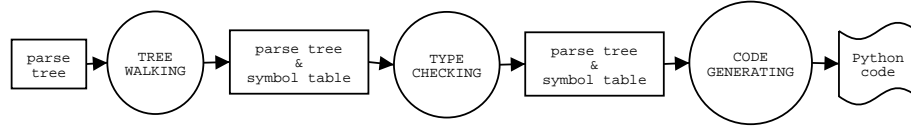


Figure 2: A traditional way to do type check and code generation

1. Lexical Analysis

The OCL code fragment is nothing more than a big string. And this string will be processed and be converted into a sequence of tokens and terminal symbols corresponding to some predefined rules with regular expression. The Lexer will be generated with Flex.

2. Syntax Analysis

Defines a series of production rules(grammar) and their corresponding actions. Compiles them with Yacc and generates a parser. Then the output of the phase1, a sequence of tokens and terminal symbols will be parsed, and an abstract syntax tree(or parse tree) will be built following the OCL concrete syntax(the grammar of the object constraint language).

3. Type checking and code generation

There are two possible ways to perform the type checking and code generation.

1.3.1 A traditional way

The usual way to do type check and code generation is showed figure 2.

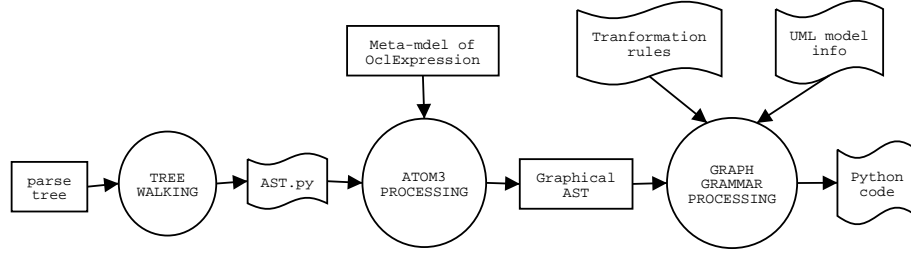


Figure 3: Type check and code generation in ATOM3

1.3.2 A new way with graph grammar

A new way to do type check and code generation in ATOM3 with graph grammar is showed in figure 3

1. Construct a meta-model in ATOM3 environment with Entity Relationship Model for OCL expression corresponding to the abstract syntax defined in the OCL2.0 specification.
2. Loads the OCL meta-model into ATOM3.
3. Loads AST.py into ATOM3, create a graphical AST.
4. Type check and code generation with graph grammar.

In this project, we followed the second path.

1.4 Why did we choose the second way

By choosing the second way, we entail more risks to the project. But we do have the following reasons:

1. By constructing the meta-model of OCL expression in ATOM3, we will have a better understanding of OCL, and it helps us to define a grammar which later could be used in yacc to build a OCL parser.
2. After the OCL expression is parsed, we will have a graphical abstract syntax tree which can be used to check errors when parsing complicated OCL constaints. Actually, it may be the best way to debug if we can visualize the parse output.
3. A transformation engine with graph grammar has already been build into ATOM3, so it will be cool to use ATOM3 old features to build up a new feature for ATOM3 itself.

4. We are not intending to build a commercial software and deliver it to market right away. Instead, we are interested in exploring different ways in which ATOM3 could be used.

2 The works I have done for this project

First, I carefully studied the OCL2.0 specification and part of the UML specification and gave a presentation on OCL. Second, I designed prototypes of meta-model of OCL expression with ER model in ATOM3 environment. Third, I implemented a series of prototypes of an OCL parser and test it with cycle test.

2.1 An OCL presentation

This presentation is on what UML object constraint language is and how OCL can be used in modeling and meta-modeling. You can refer to my presentation webpage for detailed info. Hopefully, I can update that HTML file with a nicer layout in the near future.

2.2 Prototypes of meta-model of OCL expression with ER-model

The abstract syntax of OCL is defined via UML class diagram. At the time being, ATOM3 does not support inheritance, an association between two superclasses in the UML class diagram will explode a m to n relationship in the ATOM3 ER diagram. If we build the meta-model for the whole OCL in one time, the model will easily become totally unmanageable. But my colleges are working on improving ATOM3, and hopefully in a few weeks, ATOM3 will support inheritance. So, the meta-model will also be built by prototyping.

2.2.1 Prototype 1

1. Requirements:

To build an abstract syntax tree representing the following OCL expressions:

```
--No instance of G will be named as "stop"
```

```
context G inv:
```

```
self.name <> "stop"
```

2. Design:

- A UML class diagram(figure 4)

Please note that in the UML class diagram of prototype 1, OclExpression, PropertyCallExp, ModelPropertyCallExp and LiteralExp are abstract classes.

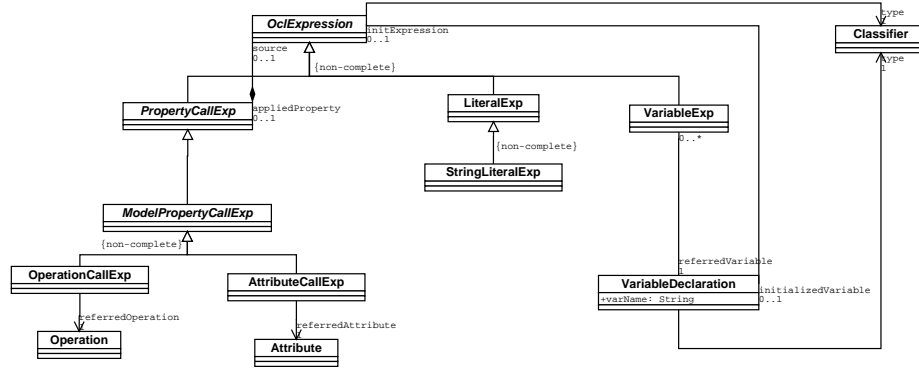


Figure 4: Class diagram of Prototype 1

- A meta-model built in ATOM3 (figure 5)
As we can see here, even for a tiny subset of OCL expression, a one-to-one association between `OclExpression` and `PropertyCallExp`, explodes to a two-to-four relation, since a `PropertyCallExp` can be either an `OperationCallExp` or an `AttributeCallExp`, and an `OclExpression` can be an `OperationCallExp`, or an `AttributeCallExp`, or a `StringLiteralExp`, or a `VariableExp`.
- An abstract syntax tree for the above example (figure 6)

2.2.2 Prototype 2

1. Requirements:

To build an abstract syntax tree representing the following OCL expressions:

--Not any two instances of type G have the same name

context G inv:

G.allInstances()->forall(e | self <> e implies e.name <> self.name)

OR:

G.allInstances()->forall(e1,e2 | e1 <> e2 implies e1.name <> e2.name)

The above is just syntactic sugar for:

```
G.allInstances()->forall(e1 | G.allInstances()
    ->forall(e2 | e1 <> e2 implies e1.name <> e2.name))
```

2. Design:

- A UML class diagram (figure 7)
- A meta-model built in ATOM3 (figure 8)
- An abstract syntax tree for the above example (figure 9)

In the ATOM3 meta-model, all the relation objects have two attributes: `roleName` and `roleName2` representing the role of the two ends of the association. Both of them may be blank in the OCL expression meta-model, but I add

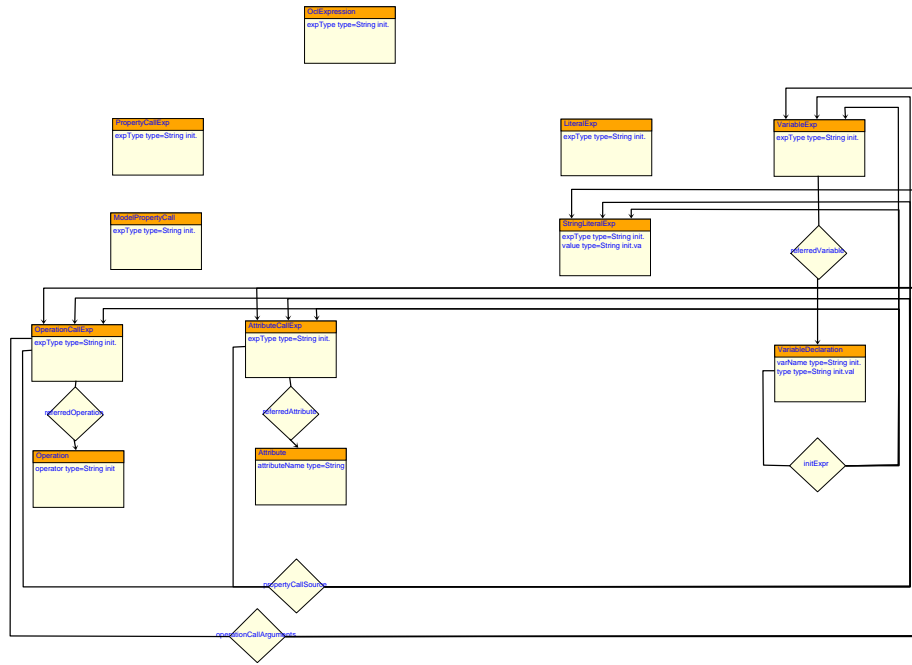


Figure 5: A meta-model of Prototype 1

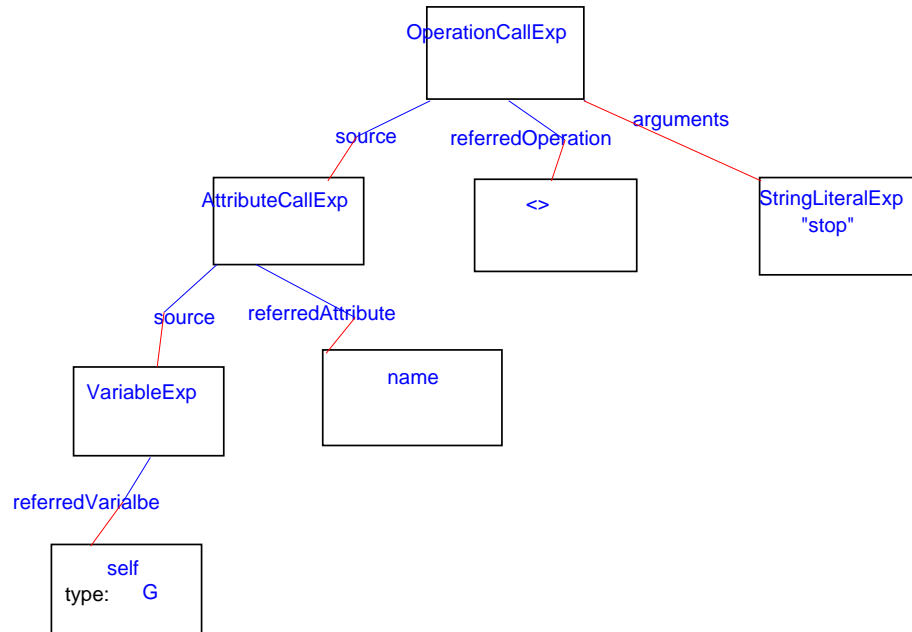


Figure 6: An abstract syntax tree for the example in Prototype 1

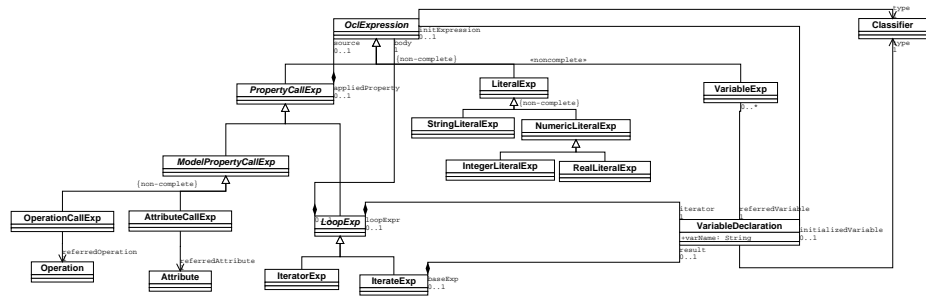


Figure 7: Class diagram of Prototype 2

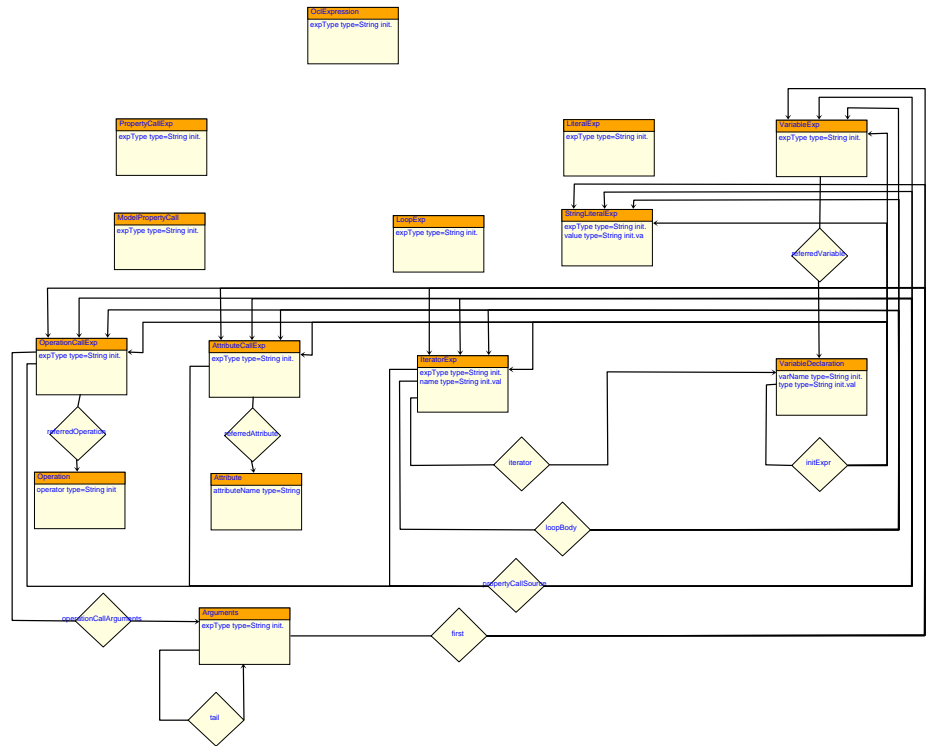
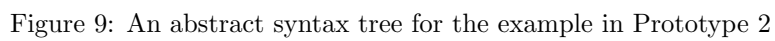


Figure 8: A meta-model of Prototype 2



them to simplify the future code generation.

According to the abstract syntax of OCL, each `OperationCallExp` has a ordered list of arguments. We have two options to satisfy this requirement. One is to add an attribute to the `Argument` entity indicating the position of the current argument in the argument list. But this makes it harder to use graph grammar to do graph transformation. So, I chose the second option, modified prototype 2 a little bit to have this new prototype.

2.3 Prototypes of an OCL parser

Since I had not got any experience in writing a parser before, I iteratively built up prototypes of my parser to work toward the goal.

2.3.1 Prototype 1

In this prototype, we implemented an OCL parser which parses an OCL expression and builds up an AST. It also outputs a python file which contains a model of the graphical appearance of the AST. This model can be loaded in ATOM3 environment.

But even a syntactically correct OCL expression may be semantically non-sense. So, further checking should and will be done via graph grammar after the abstract syntax tree is build up. All these will be done in the next prototype.

An complete OCL grammar is given in the OCL2.0 specification. But I will have to modify it for the following two reasons:

1. An abstract syntax tree corresponding to the abstract syntax given in the OCL2.0 specification should be build. And every node of the AST should be an instance of a class in the abstract syntax.
2. Some disambiguating rules must be used to disambiguate some production rules. And some of these rules can only be applied with the acknowledgements of the UML model infomation. So far, how these info should be accessed is not decided yet. We will postponed this question until Prof. Juan de Lara, the one who codes the core of ATOM3, make the decision.

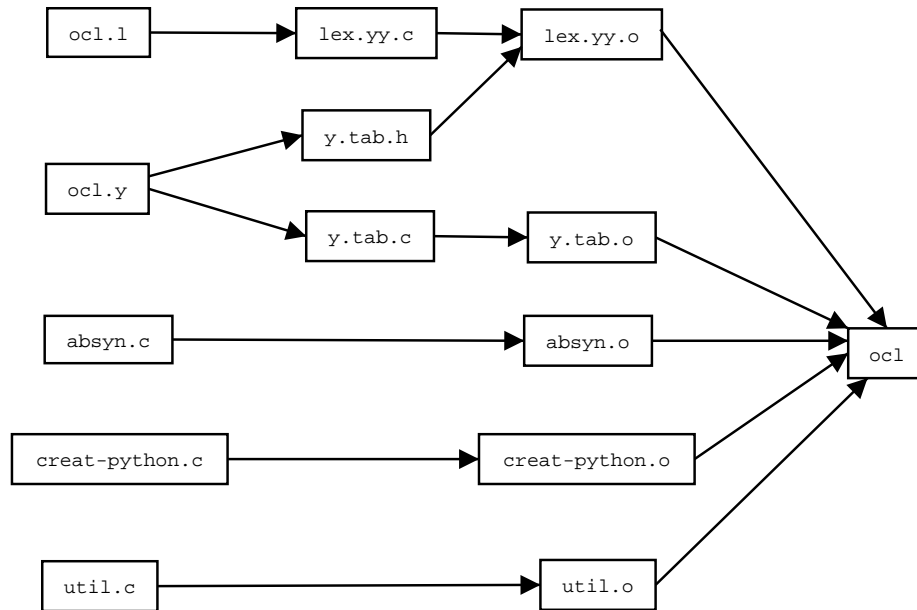


Figure 10: Dependency chart of prototype1 of the OCL parser

figure 10 shows the dependency of all the modules that compose prototype1 of the OCL parser. In the dependency chart:

ocl.l The lex specification.

ocl.y The yacc specification.

absyn.c The C module which contains the data structures and routines that build and manipulate the abstract syntax tree.

creat-python.c The C module which contains the routines that generate the graphical AST model.

util.c The C module which contains some helper routines.

Since we have only a small subset of OCL, we can only generate graphical AST of OCL expression in that subset. But this prototype can actually parse the whole set of OCL. For debugging reason, the file "screen.out" which contains info on how the OCL source is parsed.

Here is some OCL constraints examples:

```

-- This is the first example of ocl expression
inv: self.name <> "stop"

-- Second

```

```

inv: not (self.mode = "idle")

-- Third
-- Not handled yet
-- The source of an OperationCall must be specified explicitly
-- inv: name() <> "stop"

-- Fourth
inv: a+b*c>d and c<>d implies (a+d)/e>c-a

```

After we parsed the above OCL constraints, we had an AST as showed in figure 11.

2.3.2 Prototype 2

In this prototype, the parser will get input from a string rather than from a file. And we will also interface Python and C language.

Since the ultimate goal is to integrate this OCL2Python translator into ATOM3, and the core of ATOM3 is implemented in Python, the parser must be able to be called from within Python. But the parser generated with Flex and Yacc is in C language. So, we must be able to extend Python with C modules. On the other hand, we may like to embed in C some Python callbacks which will be called during the parse of OCL constraints. Fortunately, the Python interpreter is also written in C, thus with some extra efforts, we can jump back and forth between C and Python.

2.3.3 Prototype 3

In the previous prototypes of the parser, the abstract syntax tree is implemented in C. But it is more natural to implement the abstract syntax tree in an object-oriented language because the syntax itself is defined in UML class diagram. And the tree will also be easier to traverse. So, we implemented a Python module containing the data structures of the AST, and this module(absyn.py) replaces the C module(absyn.c) in prototype1.

In this prototype, we also tested the parser with cycle test.

1. We feed the OCL constraints to the parser which then output an abstract syntax tree
2. We traversed this tree and regenerated the OCL constraints with the info stored in the AST
3. Then we take the generated OCL constraints as input and loop back to step one

4. We will looped over the above steps ten times and compared the final output OCL constant with the original one

Here below is some sample codes I used in my test:

```
print '-----test complicated Ocl expression with cycle test-----'
print '## test case1: OclExpression with letExp and ifExp'
print "The oclSrc"
oclSrc = "\
inv:\n\
Let signal:Signal = typedAst.target.type.lookupSignal(sgn) in ( \n\
if signal->isEmpty() then \n\
    typedAst.sentSignal->isEmpty() \n\
else\
    (typedAst.sentSignal->notEmpty() and \n\
    typedAst.sentSignal.signal = signal) \n\
endif) "
print oclSrc
print
for i in range(10):
    a = ocl_Parse(oclSrc)
    oclSrc = "inv:\n" + a.traverse()
print "The output after 10 passes"
print oclSrc
print
```

And the output after we run the above test:

```
-----test complicated Ocl expression with cycle test-----
## test case1: OclExpression with letExp and ifExp
The oclSrc
inv:
Let signal:Signal = typedAst.target.type.lookupSignal(sgn) in (
if signal->isEmpty() then
    typedAst.sentSignal->isEmpty()
else    (typedAst.sentSignal->notEmpty() and
    typedAst.sentSignal.signal = signal)
endif)

The output after 10 passes
inv:
Let signal:Signal=typedAst.target.type.lookupSignal(sgn) in (
if (signal->isEmpty()) then
    typedAst.sentSignal->isEmpty()
else
    (typedAst.sentSignal->notEmpty() and (typedAst.sentSignal.signal = signal))
endif
```


)

Note: The test should be re-written with PyUnit. I just didn't have enough time to carefully construct the expected output string of OCL constraints.

3 Conclusion and future works

Type check and code generation should be done next. Unfortunately, ATOM3 environment does not currently support inheritance and ATOM3 is strongly typed. It would be a nightmare to design the transformation rules for the type check and code generation even we had the logic to do so in our minds. So, we may not do type check and code generation with graph grammar until the next version of ATOM3 which hopefully will support inheritance. Before that, we will just write pure Python methods to traverse the AST, build up symbol table and perform type check and code generation.

Since the OCL constraints does not stand alone, they are always stick to some models or meta-models we are building, the biggest issue for type checking will be how to retrieve info from these models. Currently, all the models built in ATOM3 will be sitting in some Python files. To get info from a model, it requires ATOM3 to load the model at run time and query its run time environment. To make it easy to do, I will discuss with Prof. Juan de Lara, and develop an API for achieving info from models.

But even we did not complete writing the whole OCL2Python translator, the experiences we gained in parsing the OCL constraints and creating an ATOM3 model of the AST(the graphical AST) can be ported to some other projects, such as processing a set of differential equations. For this instance, we build a meta-model in ATOM3 for differential equations, then we can graphically build a model for each differential equation and use constant folding and other mechanisms to transform these graphical models. But manually building models for, say, a hundred differential equations could be quite painful and time consuming. So, we can apply the parsing techniques and everything will be done quickly and automatically.

4 Acknowledgements

Great thanks to Prof. Hans Vangheluwe for his thorough guidance throughout the project.

I also want to thank all MSDL members for their excellent presentations on modelling and simulation. I really learn a lot from these guys.

References

- [1] esponse to the UML 2.0 OCL RfP (ad/2000-09-03), Revised Submission, Version 1.3
- [2] MG Unified Modeling Language Specification, Version 1.4
- [3] lex & yacc”, by John R. Levine, Tony Mason and Doug Brown, O’Reilly
- [4] Programming Python”, by Mark Lutz, O’Reilly
- [5] Compilers Principles, Techniques, and Tools”, by Alfred V.Aho, Ravi Sethi and Jeffrey D.Ullman
- [6] Modern Compiler implementation in C”, by Andrew W.Appel