An Overview of F-OML: An F-Logic Based Object Modeling Language

Mira Balaban
Computer Science Department
Ben-Gurion University
Beer-Sheva, Israel
mira@cs.bgu.ac.il

Michael Kifer
Department of Computer Science
Stony Brook University
NY 11794-4400, USA
kifer@cs.sunysb.edu

1 Introduction

F-OML is an F-Logic based Object Modeling Language. It can be used for *extending* UML diagrams, *reasoning* about them, *testing* UML models, and defining their *syntax* (*meta-modeling*) and *semantics*. This wide range of applications of F-OML stems from several language features, including polymorphism, multi-level object modeling, and model instantiation. F-OML is layered on top of an elegant formal language of *guarded path expressions*, called *PathLP*, which is used to define objects and their types. PathLP is a logic programming language, inspired by F-logic [1]. It supports *path expressions*, *rules*, *constraints*, and *queries*, and it is easy to implement by translation into a tabling Prolog engine, such as XSB.

In this short overview we informally describe the main constructs of PathLP and F-OML, and provide examples that demonstrate the four modes of F-OML usage. Formal definitions and additional details are found in the full paper. Finally, we analyze how language features contribute to its expressiveness, and provide a brief comparison with OCL [2].

2 PathLP – the Underlying Logic of F-OML

In this section we will informally describe some of the key aspects of PathLP, the underlying modeling language of F-OML.

Path expressions. Path expressions generalize path expressions in traditional imperative object-oriented languages. They extend a similar notion in XSQL [3] and more or less correspond to path expressions in F-logic [1] systems FLORID and FLORA-2 [4]. Since PathLP path expressions contain variables, they also generalize many aspects of XPath.

A path expression consists of constants and variables (symbols prefixed with "?"), and is constructed with the operators "." and "!", and guards, written within square brackets. Examples of PathLP path expressions are shown in Table 1.

Facts, rules, queries, and constraints. Facts specify assertions, rules specify implications, and constraints restrict the legal states. Queries trigger reasoning. Here are some examples.

```
John.spouse[Mary].   John.children[Bob].   John.children[Bill].
```

Facts: `John` *has a spouse represented by the object* `Mary`*, and* `Mary` *has children* `Bob` *and* `Bill` *(and possibly others).*

Expression	Informal meaning
`Mary.spouse.age(2010)`	the age at 2010 of the spouse of Mary
`?C.student[?S].name`	given a binding `c` for the variable `?C`, binds ?S to an object who is a student of c, and returns its name
`John.child(Mary)[?C:Student,?C.age(2010)<20].name`	the name of a child of `John` and `Mary`, who is a student, whose age in 2010 is less than 20
`Person!spouse[Person]{0..1}`	defines the type of the spouse property of `Person`, restricted to be `Person` (or its subclass), and having the cardinality `0..1`

Table 1. Examples of path expressions

```
Committee::Group.      Teaching_committee:Committee.
CS_teaching_committee::Teaching_committee.  Bob:CS_teaching_committee.
```

Class hierarchy and membership assertions: `Teaching_committee` *is a member of Committee, which is a subclass of Group. Bob is a member of* `CS_teaching_committee`*, which is a subclass of* `Teaching_committee`.

```
Person!spouse[Person]{0..1}.
```

A type assertion: the type of the spouse property of Person is Person, or one of its subclasses, and the cardinality constraint is `{0..1}`.

```
?A:advisor :- ?T:Thesis, ?T.author.advisor[?A,?A:Professor].read[?T].
```

A rule stating that `?A` *is an advisor if* `?A` *has read a thesis* `?T` *of an author that* `?A` *advises.*

```
?A:good_advisor :- ?A:Professor, not ?A:mediocre_advisor.
```

A rule defining good advisors – using negation.

```
!- ?P:Professor, not ?P.degree[PhD].
```

A constraint that forbids states with a professor `?P` *that does not have a* `PhD` *degree.*

3 F-OML – The Semantic Layer

F-OML is a semantic layer on top of PathLP. It provides definitions for the various UML concepts such as *classes* and *properties* as well as a library of class and property *constructors* and *definitions*. The latter are characterized using the polymorphic expressions feature. Some of these definitions are shown below.

1. **Class construction using *Set operations* – Class intersection**.
   `intersection(?C1,?C2):Class :- ?C1:Class, ?C2:Class.`
   `?o:intersection(?C1,?C2) :- ?o:?C1, ?o:?C2.`
   `Class` is a meta-class supported by F-OML; `intersection` is a polymorphic class constructor. Different bindings for `?C1,?C2` in `?o:intersection(?C1,?C2)` define different classes. Note the multilevel modeling of `intersection(?C1,?C2)`: It is, both, a member of the meta-class `Class`, and a class having its own members.
2. **Property conjunction**. `Property` is a meta-class supported by F-OML.
   `and(?p1,?p2):Property :- ?p1:Property, ?p2:Property.`
   `?o.and(?p1,?p2)[?v] :- ?o.?p1[?v], ?o.?p2.[?v].`
3. **Inverse properties**. Example: `child = or(inverse(father),inverse(mother))`.
   `inverse(?p):Property :- ?p:Property.`
   `?o1.inverse(?p)[?o2] :- ?o2.?p[?o1].`
4. **Binary composition**.
   `compose(?p1,?p2):Property :- ?p1:Property,?p2:Property.`
   `?o.compose(?p1,?p2)[?v] :- ?o.?p1.?p2[?v].`
5. **Transitive closure**. Example: `closure(flight)`.
   `closure(?p):Property :- ?p:Property.`
   `?o.closure(?p)[?v] :- ?o.?p[?v].`
   `?o.closure(?p)[?v] :- ?o.?p.closure(?p)[?v].`
6. **Property reification**: `reif(?p):Class :- ?p:Property.`
   `(?o1,?o2):reif(?p) :- ?o1.?p[?o2].`
   `?o1.?p[?o2] :- (?o1,?o2):reif(?p).`

In addition, F-OML defines a wide variety of classes and properties, including *injective, surjective, bijective* [5], *acyclic* and *unary* properties, a *subproperty* relation, *disjoint classes, singleton classes*, and more.

```
injective(?p) :- ?p:Property, ?p.target[?T], ?T!inverse(?p){0..1}.
surjective(?p) :- ?p:Property, ?p.target[?T], ?T!inverse(?p){1..*}.
```

4 Using F-OML

We envision four modes of using F-OML: (1) Extending UML diagrams; (2) Reasoning about UML diagrams; (3) Testing UML models; (4) UML definition – meta-modeling (including syntax and semantics). The three language features that enable this versatile usage are: (1) *polymorphic expressions*; (2) *multiple level* object modeling (3) *model instantiation*. Polymorphism is enabled by parameterized expressions and by class hierarchy. Parameterized expressions function like polymorphic types in functional languages and like Java generics or C++ templates. Class hierarchy yields partial ordering over types. Multilevel modeling is enabled by the subclass partial ordering "::" and the membership relation ":". By model instantiation we mean the ability to populate classes with objects, properties with appropriate binary relations, as well as giving values to other relationships, such as subclass. Model instantiation is a key enabler of *reasoning* in F-OML, which includes model testing and querying.

Diagram extension: Figure 1 is a class diagram that models User-Table permissions in a database.

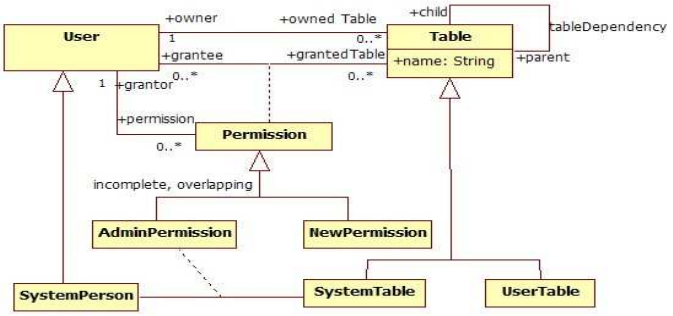

Fig. 1. A Class Diagram Example

Example 1. The owner of a table is automatically granted an access permission, and is the grantor for that permission.

```
?t.grantee[?u] :- ?t:Table, ?t.owner[?u].
?p.grantor[?u] :- ?p:Permission, ?p.grantee(Permission)[?u],
                  ?p.Table(Permission).owner[?u].
```

The above rules define the relationships `grantee` and `grantor` of an access permission to a table. The statement `?t:Table` says that `?t` represents some member of the class `Table` and `?t.owner[?u]` says that `?u` is an object that is an owner of `?t`. In the second rule, `grantee(Permission)` and `Table(Permission)` are F-OML *parametrized properties* (directed associations), defined for association classes. They map permissions to their user and table components.

Example 2. Only systems people are granted access to system tables, and there must be at least two grantees.

```
SystemTable!grantee[SystemPerson]{2..*}.
```

This is a *typing fact*. It consists of a *type path expression*, that imposes a type and cardinality constraint on all members of the `SystemTable` class.

Example 3. Tables with a common owner are linked via `tableDependency`*, i.e., via the* `parent-child` *relationship.*

```
?t.or(closure(parent),closure(child))[?s] :-
                        ?t:Table,?s:Table,?t.owner=?s.owner.
```

This example demonstrates the expressivity of the polymorphic property constructors `closure` and `or`.

Reasoning: Model querying is a major form of reasoning that plays an essential role in the process of software development, explanation, understanding, and validation. It relies on meta-modeling and uses the multilevel modeling capability.

Example 4. Find the classes related to class `User`*, and their relevant roles.*

```
?- ?a:Association, ?a.property[?p].source[User], ?p.target[?C].
```

The symbol `?-` indicates that the above statement is a query. The answers to this query are all (and only) relevant properties (roles) and their classes.

Example 5. Find all classes accessible from `User`*, and the sequence of properties in the access path:* `?- ?User!path(?path)[?C].`

`path(list)` is a parametrized F-OML property. An answer example:
`?path=[ownedTable,grantee,permission], ?C=Permission`.

Model testing: Testing is made possible due to the ability to instantiate F-OML models, i.e., to construct model states (like object diagrams).

Example 6. An illegal state: A non-owner access permission granted to self.

```
u:User.              u.grantedTable[t].Permission(Table)[p].
t.owner[v].          u.Permission(grantee)[p].grantor[u].
```

The a test might indicate that a relevant constraint has been overlooked.

Meta-modeling:

Example 7. A meta-level definition of a key attribute and a definition of an attribute named "ID" as a key attribute.

```
key(?class, ?id) :- ?class:Class, ?class.attribute[?id].name["ID"].
!- key(?class, ?att), ?o1:?class, ?o2:?class,
       ?o1.?att[?val1], ?o2.?att[?val2], ?val1 = ?val2.
```

Example 8. An association having cardinality constraint 1 at one end, is an ownership association for the other end. In Figure 1 the properties `owner` and `grantor` are ownership properties.

```
ownership(?p) :- ?p:Property, ?p.source!?p{1..1}.
```

`?p.source` stands for the source class of the property denoted by `?p`, the type path `?p.source!?p` denotes its target class, and `?p.source!?p{1..1}` restricts the cardinality of `?p` to be {1..1}.

5 Evaluation

F-OML has a number of advantages over OCL. These include broader scope (bridging model layers, pattern specification, reasoning) and applicability (testing). In particular: (1) F-OML collection manipulation yields simpler expressions; (2) F-OML supports hierarchical data structures; (3) Polymorphic expressions can express patterns; (4) Multilevel modeling is enabled by the subclass and membership relations. F-OML can express UML diagrams and their constraints, yielding powerful meta-modeling, that includes specification of syntax and semantics; (5) F-OML supports reasoning, including model querying and testing, through model instantiation.

F-OML can be shielded from the naive user by a less technical syntactic layer—similarly to how predicate logic is shielded from the user by SQL.

References

[1] Kifer, M., Lausen, G., Wu, J.: Logical foundations of object-oriented and frame-based languages. Journal of ACM **42** (1995) 741–843

[2] Object Management Group: UML 2.0 Object Constraint Language Specification. (2006)

[3] Kifer, M., Kim, W., Sagiv, Y.: Querying object-oriented databases. In: ACM SIGMOD Conference on Management of Data, New York, ACM (1992) 393–402

[4] Kifer, M.: FLORA-2: An object-oriented knowledge base language. (The FLORA-2 Web Site) http://flora.sourceforge.net.

[5] Wahler, M., Basin, D., Brucker, A., Koehler, J.: Efficient analysis of pattern-based constraint specification. In: Software and Systems Modeling. (2009)