Logic-Based Model-Level Software Development with F-OML

Mira Balaban[1,⋆] and Michael Kifer[2,⋆⋆]

[1] Ben-Gurion University, Israel
`mira@cs.bgu.ac.il`
[2] Stony Brook University, USA
`kifer@cs.sunysb.edu`

Abstract. Models are at the heart of the emerging *Model-driven Engineering* (*MDE*) approach in which software is developed by repeated transformations of models. Intensive efforts in the modeling community in the past two decades have produced an impressive variety of tool support for models. Nonetheless, models are still not widely used throughout the software evolution life cycle and, in many cases, they are neglected in later stages of software development. To make models more useful, one needs a powerful model-level IDE that supports a wide range of object modeling tasks. Such IDEs must have a consistent formal foundation.

This paper introduces *F-OML*, a language intended as an expressive, executable formal basis for model-level IDEs. F-OML supports a wide variety of model-level activities, such as *extending* UML diagrams, defining *design patterns*, *reasoning* about UML diagrams, *testing* UML diagrams, specification of *Domain Specific Modeling Languages*, and *meta-modeling.* F-OML is a semantic layer on top of an elegant logic programming language of *guarded path expressions*, called *PathLP*. We believe that a combination of current object technology with F-OML as an underlying language can lay the basis for a powerful model-level IDE.

1 Introduction

Models are at the heart of the emerging *Model-driven Engineering* (*MDE*) approach in which software is developed by repeated transformations of models. The MDE approach is motivated by the understanding that the growing complexity of software requires multiple levels of abstraction that programming languages do not usually support [1].

Intensive efforts in the modeling community in the last two decades have produced an impressive variety of tool support for models. Nevertheless, models are still not widely used throughout the software evolution life cycle and, in many cases, they are neglected in later stages of software development. Moreover, users neglect specification of essential constraints, since they are not supported by the software tools that implement the models. To make models more useful, one

⋆ Supported in part by the Paul Ivanir Center for Robotics and Production Management at Ben-Gurion University of the Negev.

⋆⋆ Supported in part by the NSF grant 0964196.

needs a powerful model-level IDE that supports a wide range of object modeling tasks. Such IDEs must have a consistent formal foundation.

This paper[1] introduces *F-OML*, a language intended as an expressive, executable formal basis for model-level IDEs. F-OML can support a wide variety of model-level activities, such as *extending* UML diagrams, defining *design patterns*, *reasoning* about UML diagrams, *testing* UML diagrams, specification of *Domain Specific Modeling Languages* (*DSMLs*), and *meta-modeling.* F-OML provides a formal API for object modeling, supported by a well-defined semantics and a provably correct execution methods. The visual models (e.g., UML) provide concrete syntax on top of the language abstract syntax.

F-OML is a semantic layer 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 [3]. It supports *path expressions*, *rules*, *constraints*, and *queries*, and can be easily implemented in a tabling Prolog engine, such as XSB. PathLP has three distinctive features that make it a particularly powerful tool for object modeling: (1) polymorphism of language expressions and of class hierarchies; (2) multilevel object modeling; (3) executable model instantiation. F-OML consists of the two first-class object concepts of *Class* and *Property*, and a library of parameterized constructors and features. The paper defines PathLP and F-OML, and illustrates them with examples of various model-level tasks.

Section 2 describes F-OML by example, and Section 3 formally introduces the PathLP language. The F-OML layer is described in Section 4, and its usage is demonstrated in Section 5. Section 6 briefly describes related work and Section 7 concludes the paper.

2 F-OML by Example

2.1 PathLP Introduction

PathLP consists of *path expressions, facts, rules, queries* and *constraints.*

Path Expressions: The key syntactic element of PathLP, which generalizes path expressions in traditional object-oriented languages is *path expression.* They extend a similar notion in XSQL [4], an F-logic [3] based language for querying object-oriented databases, in the direction of the more general path expressions in the F-logic systems [5]. PathLP also generalizes many aspects of XPath.

The building blocks of path expressions are *terms, guards, cardinalities*, and two *operators*: "." and "!". Terms are constructed from *constant symbols* and *variables* (which are denoted by symbols prefixed with "?"). Guards are path expressions written within square brackets. Examples of PathLP path expressions are shown in Table 1. In these path expressions, `Mary, spouse, ageAt(2010)`, and `?C` are terms, `[?S]` and `[Person]` are guards, and `{0..1}` is a cardinality. `?C:Student` and `?C.ageAt(2010)<20` are query formulas.

[1] A preliminary overview on this work appeared in [2].

Intuitively, the "." operator provides navigation along *value paths*. Therefore, in a path expression $n.e_1. \; \ldots \; .e_k$ (ignoring guards), we refer to n as a *node* and to the e_i-s as edges. The "." in $n.e$ yields a "value" that results from navigation along an "edge" e whose origin is n. There can be multiple such edges as, for example, in `John.childOf`.

Table 1. Examples of path expressions

Expression	Informal meaning
`Mary.spouse.ageAt(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.childWith(Mary)[?C].name, ?C:Student, ?C.ageAt(2010)<20`	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}`	restricts the type of the `spouse` property of `Person` to be `Person`, and to have cardinality `0..1`

The intuition behind the operator "!" is similar to ".", but "!" yields a *type* of an edge, rather than its value. For instance, `Person!spouse` denotes the possible types of a `spouse` edge of a `person`, `Person!spouse[Person]` checks that `Person` is one of these types (implying that so are also all of its super types), and `Student!thesis[Document]!length[NaturalNumber]` checks that the type of a `Student thesis` is `Document`, and the type of the `length` edge from `Document` is `NaturalNumber`. Type path expressions can also have constrained cardinalities of the form {`low..high`}, which specify the minimum and maximum cardinality for member nodes (precise definition in Section 3). Altogether, the semantic domain of PathLP can be viewed as directed *value graphs* and directed *type graphs* sharing the same set of nodes.

Guards play the role of *selectors*. They are usually variables or constants. For instance, `John.childOf[?X].name` binds the variable `?X` to the object that represents one of John's children, and denotes the value of the edge labeled `name` of that object. Similarly, `John.childOf[Mary].height` checks that Mary is one of the children of John and denotes her height. Guards can be followed by query formulas that act like tests on the intermediate values of path expressions. For instance, `John.childOf[?X].name,?X:Student,?X.ageAt(2009)<10`, binds `?X` to an object that represents one of Johns children who is a student and is under 10 years old, and denotes the name of that child.

Facts, Rules, Queries, and Constraints: Facts specify assertions, rules specify implications, and constraints restrict the legal states, by specifying forbidden states. Queries trigger reasoning.

Fact Examples

1. `John.spouse[Mary].   John.childOf[Bob].   John.childOf[Bill].`
   `John` has a spouse `Mary`, and children `Bob` and `Bill` (and possibly others).
2. Inclusion and membership assertions: Nodes are related by two relations "::" and ":" that have properties of set inclusion and membership, respectively.

`Bob:CS_committee.  CS_committee::Academic_committee.`
`Academic_committee:Committee.  Committee::Group.`

The intuition behind these facts is
`Bob` $\in$ `CS_committee` $\subseteq$ `Academic_committee` $\in$ `Committee` $\subseteq$ `Group`.

3. A type assertion: `Person!spouse[Person]{0..1}`.
   `Person` is one of the types of the spouse edge of `Person`, and its cardinality constraint is `{0..1}`.

Rule and Constraint Examples

1. `?S.studentOf[?Prof] :- ?S:Student, ?S.takes.teaches[?Prof].`
   A rule stating that if `?S` is a member of the `Student` node and `?S` takes a course taught by `?Prof` then `?Prof` is a value of a `studentOf` edge from `?S`.
2. `?A:advisor :- ?T:Thesis, ?T.author.advisor[?A].read[?T],`
   `?A:Professor.`
   This rule states that `?A` is an advisor if `?A` has read a thesis `?T` of an author that `?A` advises.
3. `!- ?P:Professor, not ?P.degree[PhD].`
   A constraint that forbids states where a professor `?P` has no `PhD` degree.

2.2 Introduction to F-OML

F-OML uses PathLP for formulating the two fundamental object oriented concepts of *Class* and *Property*. This approach is close to meta-modeling semantics [6], since the two model levels are expressed in PathLP, which defines the abstract syntax and semantics of models. F-OML specifications are executable since they are expressed in PathLP. The library of constructors and properties is a major source of expressivity for F-OML.

The following examples use model-level constructors for expressing class invariants and for generalizing a concrete invariant into an invariant pattern. We use UML class diagrams for visualizing (as concrete syntax) F-OML expressions that specify classes, properties and cardinality constraints.

Example 1. Figure 1 describes a `User-Table` class diagram. A table has a single user as its owner, and a user might own multiple tables. The `tableDependency` association is not constrained by multiplicity constraints.

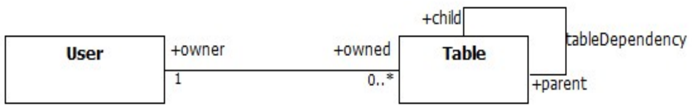

Fig. 1. User-Table ownership Class Diagram

Assume that the model requires the constraint: "Tables with a common owner are directly or indirectly linked via the `tableDependency` association." In order to express this constraint there is a need to relate a table to all of its indirect parent tables and all of its indirect child tables. The property constructor

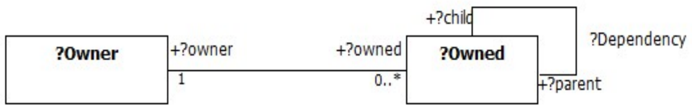

Fig. 2. Single ownership Class Diagram pattern

`closure` is used to define the new *parameterized properties* `closure(parent)` and `closure(child)` that provide the necessary mappings. The `or`-constructor defines a parameterized property which is the union mapping of its arguments. Therefore, the property `or(closure(parent),closure(child))` maps a table to all of its direct and indirect parent and child tables. The required constraint is captured by a rule stating that a table `?s` is a direct or indirect parent or child table of a table `?t` if `?t` and `?s` have the same user owner:

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

Example 2. Suppose that the above constraint is identified by domain experts as a typical ownership situation that can serve as a *reference model.* They define the following design pattern: *A single owner of multiple objects of the same class requires mutual relationships between its owned objects.* A solution is to instantiate the reference model given by the *class diagram pattern* in Figure 2 and the associated constraint pattern.

```
?o1.or(closure(?parent),closure(?child))[?o2] :-
                       ?o1:?Owned,?o2:?Owned,?o1.?owner=?o2.?owner.
```

Instantiation is performed by replacing class and property variables (`?Owned` and `?owner`) with concrete classes and properties (`Table` and `owner`) in Figure 1.

The reference model formulation exploits the expression polymorphism and the multi-level features of PathLP. Due to the executable nature of PathLP at the foundation, we can further manipulate the reference model and its instantiation.

3 PathLP — The Underlying Logic of F-OML

3.1 Syntax

The ***alphabet*** of the PathLP language includes countably many constant symbols, (e.g., Foo_123) and variables (designated with the "?" prefix, e.g., $?x$), plus the auxiliary symbols "!", ":", "::", "[", "]", "(", ")", ":-", ">", "=", and so on.

A ***term*** is defined recursively as either a variable, a constant, or an expression of the form $c(t_1, ..., t_n)$, where c is a constant and t_1, ..., t_n, $n \geq 0$, are terms. The latter kind of a term is called a ***compound term***.

Path Expressions: The following BNF productions define path expressions where `Var`, Term, `NonNegInt` denote variables, terms, and non-negative integers.

```
PathExpr        := ObjectPathExpr | TypePathExpr
ObjectPathExpr  := (Expr '.')* Expr
TypePathExpr    := (Expr '!')+ Expr [ '{' Cardinality '}' ]
Expr            := GuardedExpr | UnguardedExpr
UnguardedExpr   := Term
GuardedExpr     := UnguardedExpr '[' Guard ']'
Guard           := UnguardedPathExpr (',' UnguardedPathExpr)*
Cardinality     := (Var|NonNegInt) '..' (Var|NonNegInt|'*')
```

where `UnguardedPathExpr` is a `PathExpr` ending with `UnguardedExpr` (it is not defined explicitly to simplify the presentation).

PathLP expressions resemble those of XPath.[2] Examples include `John.spouse`, `Person!name[String]`, and `Person!spouse[Person]{0..1}`. The last two of these are ***guarded*** path expressions. The definition of query formulas, below, uses `GuardedPathExpr` as a syntactic category for guarded path expressions.

Queries and constraints: PathLP uses *query formulas* as selectors in path expressions and as bodies of PathLP inference rules and constraints.

```
Query       :=  '?-' QueryFormula '.'
Constraint  :=  '!-' QueryFormula '.'
QueryFormula      := ElementaryFormula
                   | 'not' QueryFormula | '(' QueryFormula ')'
                   | (QueryFormula ('and'|'or') QueryFormula)
ElementaryFormula := Membership|Subset|GuardedPathExpr|Comparison
Membership        := Term ':'  Term
Subset            := Term '::' Term
Comparison        := Term Op Term
Op                := '=' | '!=' | '>' | '<' | '>=' | '=<'
```

The `and` connective in query formulas can be replaced by a comma.

Facts and Rules: We introduce a new syntactic category `Consequent`, that represents formulas that are allowed as facts or rule consequences. Such formulas are considerably simpler than query formulas and even than elementary formulas – the usual restriction in logic programming languages. `Consequents` are `ElementaryFormulas` that are subject to the following restrictions:

– Comparison formulas can be only of the form `Term = Term`. That is, we are not allowed to infer facts like `a > b`.
– Path expressions can have only one operator "." or "!" and only terms as guards. That is, they can take one of the following forms: `Term.Term[Term]`, `Term!Term[Term]`, or `Term!Term[Term]{Cardinality}`.

These restrictions make PathLP reducible to Logic Programming and provide a way for an efficient implementation. Finally, the definition of facts and rules:

```
Fact        :=  Consequent '.'
Rule        :=  Consequent ':-' QueryFormula '.'
```

[2] Apart from the differences in the underlying models, PathLP variables turn it more expressive than XPath. Although PathLP expressions have no descendant-or-self wildcards of XPath, these can be defined recursively by rules.

PathLP has three language features that make it a powerful foundation for supporting object modeling:

1. **Polymorphism:** PathLP has two forms of polymorphism: *expression polymorphism*, which enables the specification of patterns and reference models as in `or(closure(?parent),closure(?child))[?o2]` – see Example 2, and the standard *class hierarchy polymorphism* of object-oriented modeling.
2. **Multi-level object modeling:** This feature enables full meta-modeling, defining the abstract syntax on the meta-model level, and the semantics on the model level, as in:

   ```
   intersection(?C1,?C2):Class :- ?C1:Class, ?C2:Class.
   ?o:intersection(?C1,?C2) :- ?o:?C1, ?o:?C2.
   ```

   The first rule specifies the class constructor `intersection` on the meta-model level, and the second rule partially specifies its semantics, on the model level. Section 4 provides further explanations.
3. **Executable language:** PathLP is an executable standalone language (unlike OCL). It supports model instantiation (*i.e.*, population of objects and links) which enables testing and querying on various modeling levels.

3.2 Semantics

The semantic domain of PathLP is a set of entities, over which various structures (value graphs, type graphs, membership and inclusion relations, and cardinality constraints) are defined. The domain does not differentiate entities by their role: node, edge, or type: the same entity can play different roles depending on the syntactic context. Formally, *up to an isomorphism*, the ***domain*** is a set of all *ground* (i.e., variable-free) terms, which includes the values of standard data types (strings, numbers, etc.).

A PathLP ***interpretation***, $\mathcal{I}$, is a tuple of the form $\langle U, I_C, I_V, I_F, I_{val}, I_{type}, I_{min}, I_{max}, \in_{\mathcal{I}}, \prec_{\mathcal{I}}\rangle$, where U is the *domain*, I_C is a mapping from constant symbols to U; I_V is a variable assignment mapping, which is a total function $Vars \longrightarrow U$; I_F is a function $U \longrightarrow (\cup_{n=0}^{\infty} U \longrightarrow U)$, which associated to every element in U a polyadic function $\cup_{n=1}^{\infty} U \longrightarrow U$; and I_{val}, I_{type} are both ternary relations over U. $I_{min}, I_{max} : U \times U \longrightarrow (Integers \cup \{*\})$ are mappings such that $0 \leq I_{min}(x, y) \leq I_{max}(x, y)$ for all $x, y \in U$. $\in_{\mathcal{I}}$ and $\prec_{\mathcal{I}}$ are binary relations over U: $\in_{\mathcal{I}}$ represents the ***membership relation***, and $\prec_{\mathcal{I}}$ is a partial order that represents the ***subset relation***.

The mapping I_{val} determines the values of edges. A triple $(n, e, v) \in I_{val}$ defines v as the value of the edge e of node n. For a given node n and edge e, there can be multiple such triples, since the value graph structure allows multiple edges with the same label for a node. The mapping I_{type} determines the types of edge values. A triple $(n, e, t) \in I_{type}$ defines t as the type of the edge e of node n. Typing should satisfy *closure properties* with respect to the subset relation, and *well-typing properties* with respect to the value mapping.

Closure Properties:

- *Upward-closure*: if $(n, e, t) \in I_{type}$ and $t \prec_{\mathcal{I}} t'$ then also $(n, e, t') \in I_{type}$ (if e has type t then every supertype of t is also a type of e).
- *Inheritance*: if $n \prec_{\mathcal{I}} n'$ and $(n', e, t) \in I_{type}$ then $(n, e, t) \in I_{type}$ (if e has type t for a node n' then it has type t for every subset-related node of n'; i.e., e is inherited).

Well-typed Interpretations: *Well-typed* interpretations, first introduced in [3], enforce well-typing of edge values of member nodes. Well typing has two aspects: A typing restriction for each value, and obeying the cardinality restrictions. Namely, for every value-triple $(n, e, v) \in I_{val}$, there is a type-triple $(n', e, t) \in I_{type}$ such that

- $n \in_{\mathcal{I}} n'$ and $v \in_{\mathcal{I}} t$
- $I_{min}(n', e) \leq \texttt{cardinality}(\{v \mid (n, e, v) \in I_{val}\}) \leq I_{max}(n', e)$

The membership and subset relations are required to satisfy these properties: $n \in_{\mathcal{I}} n'$ and $n' \prec_{\mathcal{I}} n''$ imply $n \in_{\mathcal{I}} n''$. This implies that the set of all the members of n' is a subset of the set of the members of n''. Note that the opposite does not have to hold.

The Meaning of PathLP Constructs

Given an interpretation $\mathcal{I}$, we define the notion of *satisfaction by interpretation* for PathLP query formulas, facts, rules, and constraints. We first define the *denotation mapping* associated with $\mathcal{I}$. The purpose of that mapping is to interpret path expressions as subsets of the domain of $\mathcal{I}$. It is common to use the same symbol $\mathcal{I}$ both for the interpretation and for its associated denotation mapping. The definitions of the denotation mapping and of satisfaction are inductive on the structure of the formulas and are mutually dependent.

Denotation of Path Expressions

- *Constant*: If c is a constant then $\mathcal{I}(c) = \{I_C(c)\}$.
- *Variable*: If $?x$ is variable then $\mathcal{I}(?x) = \{I_V(?x)\}$.
- *Unguarded expression*: If τ is a compound term $c(t_1, ..., t_n)$ (an unguarded expression) with zero or more arguments then:
  $\mathcal{I}(\tau) = \{I_F(I_C(c))(t'_1, ..., t'_n)\}$, where $t'_i \in \mathcal{I}(t_i)$ for $i = 1, ..., n$.

The previous three cases form the basis for the inductive definition of $\mathcal{I}(\tau)$, where τ is a path expression. The inductive part of the definition now follows.

- *Unguarded object path expression*: If τ is *objpathexp.expr*, where *objpathexp* is an object path expression and *expr* is a term then:

  $\mathcal{I}(\tau) = \{v \mid \exists n \in \mathcal{I}(objectpathexp), \exists e \in \mathcal{I}(expr), \text{ such that } (n, e, v) \in I_{val}\}$.

  Note that $\mathcal{I}(\tau)$ can be empty.
- *Guarded object path expression*: If τ is *ungobjpathexp*[*grd*], where *ungobjpathexp* is an unguarded object path expression and *grd* is a guard of the form *ungpathexp*$_1$, ..., *ungpathexp*$_n$ then:
  $\mathcal{I}(\tau) = \mathcal{I}(ungobjpathexp) \cap \mathcal{I}(ungpathexp_1) \cap \cdots \cap \mathcal{I}(ungpathexp_n)$

- *Type path expression*:
  - *Unguarded without cardinality constraint:* If τ is *tpathexp*!*expr*, where *tpathexp* is a type path expression and *expr* is an expression then:

    $\mathcal{I}(\tau) = \{v \mid \exists n \in \mathcal{I}(tpathexp),\ \exists e \in \mathcal{I}(expr)$, such that $(n, e, v) \in I_{type}\}$.
  - *Unguarded with cardinality constraint*: If τ is *tpathexp*!*expr*$\{lo..hi\}$, where *tpathexp* is a type path expression and *expr* is an expression then:

    $\mathcal{I}(\tau) = \{v \mid \exists n \in \mathcal{I}(tpathexp),\ \exists e \in \mathcal{I}(expr)$, such that $(n, e, v) \in I_{type}$ and $I_{min}(n, e) = I(lo),\ I_{max}(n, e) = I(hi)\}$.
  - *Guarded*: Similarly to guarded object path expressions.

Built-in `size` Terms: PathLP assigns special meaning to the properties `size()` and `size(prop)`, used for counting the number of objects in a class and the range size of a property. Thus, the denotation of these properties must satisfy:

- `size()`: $(n, \mathcal{I}(\mathtt{size}()), \mathtt{N}) \in \mathtt{I_{val}}$, where $n \in U$ and $N \geq 0$ is an integer, if and only if the set $\{v \mid v \in_{\mathcal{I}} n\}$ is finite and has cardinality N.
- `size(e)`: $(n, \mathcal{I}(\mathtt{size(e)}), N) \in I_{val}$, where $n \in U$ and $N \geq 0$ is an integer, if and only if the set $\{v \mid (n, \mathtt{e}, v) \in I_{val}\}$ is finite and has cardinality N.

Satisfaction by Interpretations

1. *Elementary formulas*
   - *Membership*: $\mathcal{I} \models t : s$, where t, s are terms, if and only if $\mathcal{I}(t) \in_{\mathcal{I}} \mathcal{I}(s)$.
   - *Subset*: $\mathcal{I} \models t :: s$, where t, s are terms, if and only if $\mathcal{I}(t) \prec_{\mathcal{I}} \mathcal{I}(s)$.
   - *Guarded path expression with and without cardinality constraints*: $\mathcal{I} \models p$, where p is a guarded path expression, if and only if $\mathcal{I}(p)$ is non-empty.
   - *Comparison formulas* $\mathcal{I} \models (t = s)$, where t, s are terms, iff $\mathcal{I}(t) = \mathcal{I}(s)$. Likewise, $\mathcal{I} \models t < s$, iff $\mathcal{I}(t) < \mathcal{I}(s)$. The definition of satisfaction for the remaining comparisons is similar.
2. *Query formulas:*
   - *And*: $\mathcal{I} \models t$ `and` s iff $\mathcal{I} \models t$ and $\mathcal{I} \models s$.
   - *Or*: $\mathcal{I} \models t$ `or` s iff either $\mathcal{I} \models t$ or $\mathcal{I} \models s$.
   - *Not*: $\mathcal{I} \models$ `not` t iff it is not the case that $\mathcal{I} \models t$.
3. *Rules and facts*: $\mathcal{I} \models (t$ `:-` $s)$ if and only if either $\mathcal{I} \models t$ or $\mathcal{I} \not\models s$. This also covers the case of satisfaction for PathLP facts, since we can view any fact t as a rule of the form t `:-` *true*.
4. *Constraints*: $\mathcal{I} \models$ (`!-` *queryformula*) iff $\mathcal{I} \not\models$ *queryformula*.

A PathLP interpretation that satisfies the facts, rules, and constraints of a PathLP specification is a ***model*** of that specification. As usual in logic programming, we focus on ***canonical*** models. Without negation (`not`), there is a unique least model, which is the canonical model. With negation, the semantics is defined using so-called *well-founded* models [7]. A PathLP specification is *satisfiable* if it has a canonical model. An ***answer*** to a query `?-` *queryformula* is the set of all instantiations of *queryformula* satisfied by the canonical model.

With no negation, PathLP reduces to classical logic analogously to the reduction of F-logic to classical logic [3] and is semi-decidable. With negation, it reduces to logic programs with the well-founded semantics and can be implemented on top of a tabling deductive engine, like XSB, similarly to the FLORA-2 implementation of F-logic [5]. Without function symbols, PathLP is decidable and has polynomial data complexity even with negation.

4 F-OML – The Semantic Layer over PathLP

F-OML uses PathLP to define axioms for two basic notions of object modeling, *classes*, and *properties*, along with their *interrelationships*. `Class` characterizes objects that function as collections of objects. `Property` defines objects that function as mappings among classes. The definition covers three modeling levels: the *Meta Model* level (OMG's M2 level) that specifies the abstract syntax of F-OML models, and *Model* and *Data* levels (OMG's M1 and M0 levels), that specify the semantics of F-OML specifications.

F-OML Syntax: Figure 3 presents the meta-model of F-OML notions.

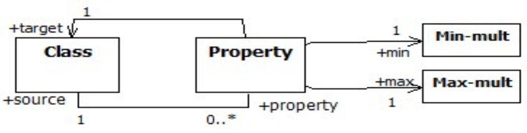

Fig. 3. Meta-model of F-OML

This meta-model is defined by the following PathLP specification:

1. *F-OML classes*, i.e., members of `Class`, have multiple properties which are members of `Property`: `Class!property[Property].`
2. F-OML *properties*, i.e., members of `Property`, have a unique source class, target class, and minimum and maximum multiplicities:
   `Property!source[Class]{1..1}.` `Property!target[Class]{1..1}.`
   `Property!min[Min_mult]{1..1}.` `Property!max[Max_mult]{1..1}.`
3. *`Class`-`Property` inter-relationships*: `Property` is a member of `Class`, and the source of a property is a class with that property:
   ```
   Property:Class.
   ?C.property[?p] :- ?p:Property,?p.?ST[?C],(?ST=source or ?ST=target).
   ?p.source[?C] :- ?C:Class, ?S.property[?p].
   ```
4. `Class` and `Property` properties are not defined on other objects:
   ```
   !- ?C.property[?p], not ?C:Class.
   !- ?p.target[?C], not?p:Property.
   ```
   Similarly for other `Property` properties.

An ***F-OML specification*** is a collection of class and property facts:

1. *Class definitions*: $\{t_i : \texttt{Class}\}_{i=1...n}$, where $t_1, \ldots, t_n$ are ground (i.e., variable-free) terms. These are the *classes* of the model.
2. *Property definition*: $\{\langle p_i.source[t_j], p_i.target[t_k], p_i.min[n_i], p_i.max[x_i]\rangle\}$ $_{i=1...m}$, where $p_1 \ldots p_m$ are all different ground terms; t_j, t_k are classes of the model; and $n_i \leq x_i$ are natural numbers, where x_i can also be $*$. The p_is are the *properties* of the model.
3. *Additional constraints*: PathLP specification imposing inter-relationships among the classes or the properties.

An *atomic F-OML specification* is one whose classes and properties are constants. A *non-atomic F-OML specification* might have classes such as `intersection(User,` Guest) or properties such as inverse(owner). Example 3 presents a (non-atomic) F-OML specification that describes the class diagram in Figure 1.

Example 3. An F-OML specification for Figure 1.

```
User:Class.  Table:Class.  owned=inverse(owner).  parent=inverse(child).
owner.source[User].  owner.target[Table].  owner.min[1].  owner.max[1]
owned.source[Table].  owned.target[User].  owned.min[1].  owned.max[1]
parent.source[Table]. parent.target[Table]. parent.min[0]. parent.max[*].
child.source[Table]. child.target[Table]. child.min[0]. child.max[*].
```

An ***F-OML pattern*** is an F-OML specification with non-ground classes or properties. F-OML patterns function as reference models for typical problems.
F-OML semantics: An ***F-OML state*** is a PathLP canonical model that satisfies axioms that define the intended meaning of F-OML classes and properties:

1. Semantics of properties of classes:
```
?C!?p[?T]{?low..?hi} :-
   ?p:Property,?p.source[?C],?p.target[?T],?p.min[?low],?p.max[?hi].
```
2. Classes must not have undeclared properties:
```
!- ?C:Class, ?C!?p[?T]{?low..?hi},
     not( ?p:Property, ?p.source[?C], ?p.target[?T],
          ?p.min[?low], ?p.max[?hi] ).
```
3. Members of classes can have only the properties declared for their classes:
```
!- ?o:?C, ?C:Class, ?o.?p[?v], not ?C!?p[?x].
```

The set of *members of a class* C in an F-OML state $\mathcal{I}$ is the set of objects that relate to it under the membership relation: $\{e | e \in_{\mathcal{I}} \mathcal{I}(\mathtt{C})\}$. Due to space limitations we omit the notions of *satisfiability* and *finite-satisfiability* in F-OML.

F-OML Specifications and Class Diagrams: An atomic F-OML specification is equivalent to a class diagram that has the same classes, properties, and multiplicity constraints. A non-atomic F-OML specification can enforce inter-relationships among classes or properties, as in GuestUser:Class; GuestUser = difference(User,RegisteredUser). Such inter-relationships are inexpressible by class diagrams.

The correspondence between F-OML specifications and class diagrams has several important consequences. First, F-OML specifications can be **visualized** by class diagrams. Second, F-OML state can be used for formulating and implementing object modeling tasks. Third, results on satisfiability [8] and finite satisfiability [9] can be used for static analysis.

Parameterized Construction and Characterization

F-OML provides specification for a wide variety of library *constructors* and *predicates* that enable definition of non-atomic F-OML specifications and F-OML patterns. Due to space restrictions, we present just a few, and provide only object-level axioms, and omit meta-level characterization.

1. **Class construction using *Set operations*:**
```
?o:intersection(?C1,?C2):- ?o:?C1, ?o:?C2.
```

2. **Finite class construction:** Defined by the `classOf` class constructor, e.g.,
   `Color = ClassOf([red, blue, yellow]).`
   `?o:ClassOf(?List) :- ?List.members[?o].`
   `!- ?o:ClassOf(?List), not ?List.members[?o].`
3. **Property construction using logic-based constructors:**
   Property disjunction: `?o.or(?p1,?p2)[?v] :- ?o.?p1[?v] or ?o.?p2[?v].`
4. **Property inversion:** `?o1.inverse(?p)[?o2] :- ?o2.?p[?o1].`
5. **Property composition:**
   Binary: `?o.compose(?p1,?p2)[?v] :- ?o.?p1.?p2[?v].`
   N-ary: `?o.path( [?p] )[?v] :- ?o.?p[?v].`
   `?o.path([?p|?path])[?v] :- ?o.?p.path(?path)[?v].`
   `where [?p|?path]` is Prolog List notation
   Transitive closure: `?o.closure(?p)[?v] :- ?o.?p[?v].`
   `?o.closure(?p)[?v] :- ?o.?p.closure(?p)[?v].`

F-OML provides a variety of library definitions that characterize classes and properties e.g., *injective, surjective, bijective* [10], *acyclic* and *unary* properties, a *subproperty* relation, and *disjoint* and *singleton classes*. For example,

1. *Injective properties*:
   `?p.kind[injective]:-?p:Property,inverse(?p).min[0],inverse(?p).max[1].`
   Assuming that the `Property` class has a `kind` property.
2. *The subproperty relation*: All `p`-mappings are also `q`-mappings:
   `?s.?q[?t]:- ?p:Property, ?q:Property, ?p.subproperty[?q], ?s.?p[?t].`
3. *An acyclic property*: `!- ?p:Property,?p.circularity[false],?o.closure(?p)[?o].`
4. *Disjoint classes*:
   `!- ?C1:Class, ?C2:Class, ?C1!=?C2, ?C1.disjointfrom[?C2], ?o:?C1, ?o:?C2.`

5 Using F-OML

This section illustrates various uses of F-OML for modeling objects.

I. Static Invariant Language: Figure 4 presents a class diagram that models User-Table access permissions in a database. A user that has an access permission to a table (its `grantor`), can grant access permission to another user (the `grantee`). Assume that the following invariant requirements are given:

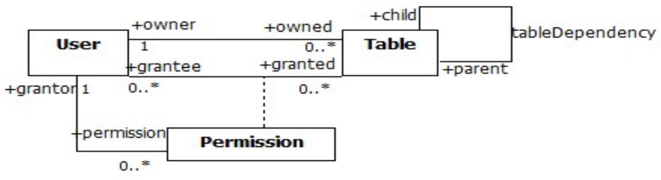

Fig. 4. User-Table permission Class Diagram

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

Requirement 2. A non-owner user cannot grant himself a permission to a table, directly or indirectly.

These requirements cannot be captured by class diagram constraints, and require a constraint language. In UML, this is provided by the *Object Constraint Language* (*OCL*) [11]. The F-OML class invariants that capture these requirements rely on the F-OML *class diagram module* (not presented in this paper) that formulates class diagram constraints. For the association class constraint, the class diagram module defines parametrized navigation properties to and from an association class to its related classes. For Figure 4, the navigation properties from a `Permission` object to its associated `User` and `Table` objects are `grantee(Permission)` and `granted(Permission)`. Requirement 1 is captured by a class diagram invariant that consists of 2 rules:

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

Requirement 2 is captured by the following rule and constraint:

```
?u.permissionGrantor(?t)[?v] :-
     ?u:User, ?u.Permission(grantee)[?p].granted(Permission)[?t],
     ?p.grantor[?v].
!- ?u:User,?t:Table,not ?u.owner[?t],?u.closure(permissionGrantor(?t))[?u].
```

The rule defines an auxiliary parametrized property `permissionGrantor(?t)` that, for a table `?t`, maps a grantee user `?u` to the grantor of his/her permission to `?t`. The rule uses the inverse navigation property `Permission(grantee)` that maps a `User`-object to the associated `Permission`-objects (this property is provided by the association class formulation in the F-OML class diagram module). The guarded path expression `?u.Permission(grantee)[?p]` selects a permission `?p` for a user `?u` and `?u.Permission(grantee)[?p].Table(Permission)[?t]` further selects the table `?t` of that permission `?p`. This constraint denies circular access granting to prevent non-owners from granting mutual access permissions.

The OCL formulation of requirement 2 is not straightforward. The rule can be captured by a similar query. However, the acyclicity constraint requires computation of a closure, which is rather complex in OCL (due to the need to compute navigation paths whose length cannot be bound a priori).

II. Design Pattern Formulation: F-OML provides natural support for formulating design patterns, including specification of their semantics. We show a design pattern generalization of the User-Table access permission model.

***Access-permission-granting* Pattern**
Problem: An access policy of *readers* to *objects* allows: (1) owner access to the owned object, (2) authorized readers granting access to object to other readers, (3) disallows granting cycles.
Solution: (1) Instantiate the class diagram pattern (a visualization of an F-OML pattern) in Figure 5. *Instantiation* means replacement of the class variables `?Reader, ?Object, ?Access` and the property variables `?owner, ?owned, ?grantee, ?granted, ?grantor, ?permission` by constants.

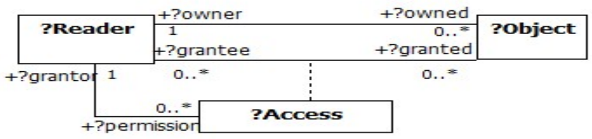

Fig. 5. Access permission Class Diagram pattern

(2) Apply the same instantiation of `Class` and `Property` typed variables to the following F-OML specification:

```
?r.accessGrantor(?o)[?q] :- ?r:?Reader,
    ?r.?Access(?grantee)[?a].?granted(?Access)[?o],?a.?grantor[?q].
!- ?r:?Reader, ?o:?Object, not ?r.?owner[?t],
           ?r.closure(?AccessGrantor(?o))[?r].
```

III. Meta-Modeling: The PathLP features of polymorphic expressions and multi-level specification enable full meta-modeling.
A ***key*** **Property:**

```
!- ?C:Class, ?C.key[?p], ?p:Property, ?o1:?C, ?o2:?C,
   ?o1.?p[?val1], ?o2.?p[?val2], ?val1 != ?val2.
```

One can postulate that a property named `ID` is a key property as follows:
`?C.key[?p] :- ?C:Class, ?C.property[?p].name[ID].`

IV. Model Query and Reasoning: Model-level reasoning has an essential role in the process of software development, explanation, understanding, and validation. F-OML supports such reasoning with PathLP queries and rules.
***Class reachability*:** In Figure 4, find all classes accessible from `User`, and the sequence of properties in the access path.

```
?C.path([?p])[?C1]:- ?C.property[?p].target[?C1].
?C.path([?p|?path])[?C1]:- ?C.property[?p].target.path(?path)[?C1].
```

The reachability query can be `?- User.path(?path)[?C]`. The answer includes `?path=[owned,grantee,permission]`, `?C=Permission`.

V. Model Testing: Model testing involves checking *mandatory* and *possible* characterizations of F-OML specifications (like object diagrams). Mandatory properties should hold in every state, and can be tested by posting F-OML queries. For example, in Figure 4, if class `Table` is restricted to be non-empty then in every state there is a `Permission` object whose `grantee` is also the owner of the table of the permission. This can be verified as follows:

```
?- ?p:Permission, ?p.grantee(Permission).owned[?T],
    ?p.granted(Permission)[?T].
```

Negative examples, that present illegal instantiations, are also helpful in model testing. A negative example can be tested by posing their negation as queries.

6 Related Work

The Object Constraint Language (OCL) [11] is the UML 2.0 language for specification of invariants, queries, and pre/post conditions on operations. It is not a standalone language; its expressions must be associated with UML diagrams. In general, the OCL handling of nested collections, unbounded data structures and recursive constraints is quite cumbersome. For example, suppose that the class `Table` in Figure 1 has two subclasses, `SystemTable` and `UserTable`, and we wish to add the invariant: "A user cannot be an owner of a system table and of a user table at the same time." The OCL formulation is:

```
Context User
inv: self.owned->select(oclIsTypeOf(SystemTable))->
     intersection(self.owned->select(oclIsTypeOf(UserTable)))->isEmpty()
```

For comparison, the F-OML 1-line formulation is:

```
!- ?u:User, ?u.owned[?st], ?st:SystemTable, ?u.owned[?ut], ?ut:UserTable.
```

F-OML has a number of advantages over OCL, including wider applicability, simplicity, full support for meta-modeling, patterns, simple management of unbounded data structures and recursion, model querying, analysis, and testing. The model analysis and the testing features rely on the status of F-OML as a standalone executable language.

Alloy [12] has been used recently for analysis, validation, and testing of UML models. Alloy is a standalone model checker, and it appears to support part of the functionality of F-OML. Yet, as a modeling language it resides at a lower level. Also, Alloy's handling of recursion and unbounded data structures like paths, cycles and tree is quite complex.

Another related work is that of [13], which extends the standard *instance diagram* language to support positive or negative examples as well as invariants. As illustrated earlier in the paper, F-OML provides an underlying logic support for the language of mandatory, possible, and negative instance diagrams.

7 Conclusion and Future Work

We presented *F-OML*, an expressive, executable modeling language, that can provide a formal basis for model-level IDEs. It is a semantic layer on top of the *PathLP* path expression language. PathLP has three distinctive features: (1) polymorphism of language expressions and of class hierarchies; (2) multilevel object modeling; (3) executable semantics. F-OML supports the basic concepts of *Class* and *Property*, and provides a library of constructors and features that function like modeling patterns.

At present, an implementation of PathLP is underway. We have already accomplished a major part of the `Class` diagram module. Once PathLP, F-OML and the class diagram module are implemented, we plan to combine it with a UML modeling tool (e.g., `http://sourceforge.net/apps/trac/mide-bgu/wiki`). Then, we can experiment with F-OML as an underlying language for the IDE, in combination with other IDE applications (`http://www.cs.bgu.ac.il/ modeling/?page_id=314`). One specifically challenging goal is extending F-OML to support dynamic models, such as statecharts or sequence diagrams.

Acknowledgments. We would like to thank Igal Khitron who implemented PathLP and provided numerous suggestions for improvements. We also thank the referees for the remarks that helped improve the presentation.

References

[1] France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: Intl. Conf. on Software Engineering, pp. 37–54 (2007)
[2] Balaban, M., Kifer, M.: An overview of F-OML: An F-Logic based object modeling language. Electronic Communications of the EASST 36 (2011)
[3] Kifer, M., Lausen, G., Wu, J.: Logical foundations of object-oriented and frame-based languages. Journal of ACM 42, 741–843 (1995)
[4] Kifer, M., Kim, W., Sagiv, Y.: Querying object-oriented databases. In: ACM SIGMOD Conf. on Management of Data, pp. 393–402. ACM, NY (1992)
[5] Kifer, M.: FLORA-2: An object-oriented knowledge base language. The FLORA-2 Web Site (2007), `http://flora.sourceforge.net`
[6] Lano, K.: UML 2 semantics and applications. Wiley Online Library, Chichester (2009)
[7] Van Gelder, A., Ross, K., Schlipf, J.: The well-founded semantics for general logic programs. Journal of ACM 38, 620–650 (1991)
[8] Berardi, D., Calvanese, D., Giacomo, D.: Reasoning on UML class diagrams. Artificial Intelligence 168, 70–118 (2005)
[9] Maraee, A., Balaban, M.: Efficient reasoning about finite satisfiability of UML class diagrams with constrained generalization sets. In: The 3rd European Conf. on Model-Driven Architecture, pp. 17–31 (2007)
[10] Wahler, M., Basin, D., Brucker, D., Koehler, K.: Efficient analysis of pattern-based constraint specifications. Software and Systems Modeling 9, 225–255 (2010)
[11] Object Management Group: UML 2.0 Object Constraint Language Specification (2006)
[12] Jackson, D.: Alloy: A new technology for software modelling. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 175–192. Springer, Heidelberg (2002)
[13] Maoz, S., Ringert, J.O., Rumpe, B.: Modal Object Diagrams. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 281–305. Springer, Heidelberg (2011)