An Overview of FOML 1.2

Mira Balaban[1], Igal Khitron[1], Michael Kifer[2]

[1] Computer Science Department, Ben-Gurion University of the Negev, ISRAEL
mira@cs.bgu.ac.il, khitron@cs.bgu.ac.il
[2] Department of Computer Science, Stony Brook University, NY, USA
kifer@cs.stonybrook.edu

FOML [1] is an expressive logic rule language that provides executable formal basis for software models. It naturally supports model-level activities, such as constraints (extending UML diagrams), dynamic compositional modeling (intensional, transformational), analysis and reasoning about models, model testing, design pattern modeling, specification of Domain Specific Modeling Languages, and meta-modeling. Meta-modeling in FOML relies on uniform treatment of types and instances and spans both definition of abstract syntax and semantics. As an executable modeling language, FOML can express and reason about multiple crosscutting multilevel dimensions.

Technically, FOML is a semantic layer on top of a compact logic rule language of *guarded path expressions*, called *PathLP*[2], an adaptation of a subset of F-logic [3]. In the overall schema of things, PathLP provides reasoning services over unrestricted instance-of and subtype relations, and over typed object-link relations, while FOML [4] provides the modeling framework. The PathLP language consists of *membership* and *subtype* expressions, *path expressions* for objects and for types, *rules*, *constraints*, and *queries*, and is implemented on top of the XSB logic reasoning engine.

This document describes how to use FOML 1.2 for specification of and reasoning about models. All other modeling capabilities derive from the combination of FOML and its underlying PathLP language.

1 Meta-modeling Terminology of FOML 1.2

FOML provides a small class and object model that is intended as a meta-meta-model for models. It can also function as a meta-model for a restricted class model. The meta-modeling capabilities of FOML are as follows:

1. **Extensional model specification:** Explicit, concrete specification of meta-models (i.e., modeling languages), including their semantics. The semantics is defined using PathLP rules and constraints.
2. **Querying and reasoning:** about defined (extensional) models and their instantiations.
3. **FOML operations:** FOML provides a set of operations on model elements. These operations define *intensional model elements*, i.e., elements that are not explicitly defined, but are constructed out of by other elements. An example is composition of properties that are explicitly declared in a class model. These operations enable powerful reasoning about model elements that are not explicitly declared.

FOML supports the following meta level terminology:

– *Class*
– *Association*
– *Property*, i.e., association-end, or role.
– *Attribute*
– *Multiplicity*
– *Class hierarchy*
– *Object*
– *Link*

These meta-elements are *members* of the metameta-class `MM_Class`.

FOML formalizes the inter-relationships among elements of these meta-classes, following the semantics of UML class models [5].

Notation: The conventions in syntax specifications are:

1. An italicized all-capital symbol denotes a PathLP constant or a variable of any type. For example, *CLASS* stands for a constant that represents a class name, or for a variable.
2. FOML keywords are written in bold.

2 Specification of a Restricted Subset of the UML Class Model in FOML 1.2

FOML supports a restricted subset of the UML Class Model. With these meta-modeling capabilities one can define restricted class diagrams, or modeling languages, like Feature models or Statecharts. In this section we show how to specify class models (diagrams). FOML meta-modeling specification is demonstrated using the class diagram in Figure 1.

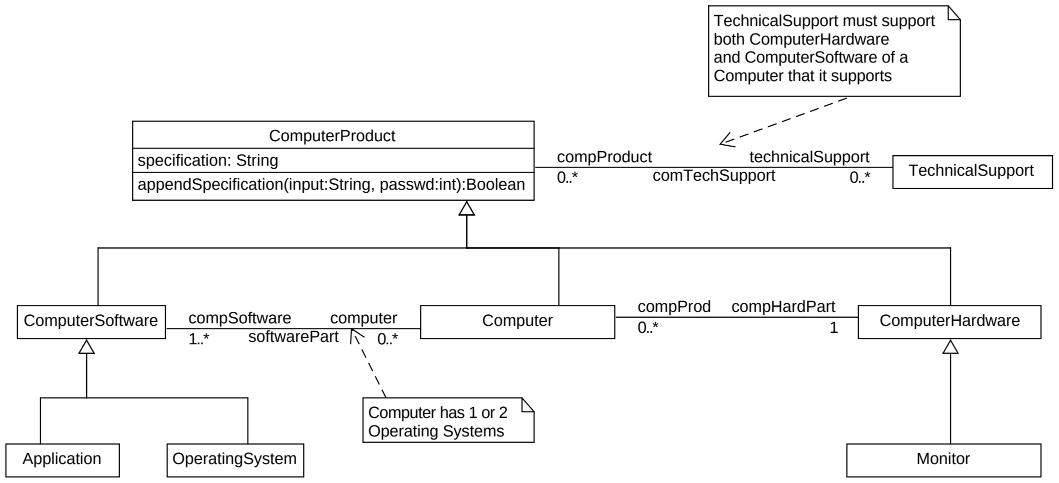

Fig. 1. A class diagram example

– *Class declaration*: *CLASS*:`Class`
  The specification of the classes in Figure 1 is given below:

```
ComputerProduct:Class;
TechnicalSupport:Class;
ComputerSoftware:Class;
ComputerHardware:Class;
Computer:Class;
Application:Class;
OperatingSystem:Class;
Monitor:Class;
```

– *Association declaration*: An association carries properties (association ends), multiplicities and end classes. All can be declared together. There are two ways to declare an association and its information. They differ in the way this information is organized. Association declaration is explained with respect to Figure 2.
  1. The first and the recommended one uses the **prop** keyword in order to declare the properties associated with the association:
     assoc_a.**prop**(*p1*, *m1*, *M1*)[*C1*]
     assoc_a.**prop**(*p2*, *m2*, *M2*)[*C2*]
     For example,

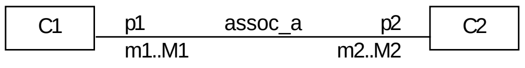

Fig. 2. An association and its associated properties, multiplicities and classes

```
softwarePart.prop(computer,0,*)[Computer];
softwarePart.prop(compSoftware,1,1)[ComputerSoftware];
```

2. The second way should be used when association name is missing and association is defined implicitly by its inverse properties.
   p1.**classes**(*c2*, *m1*, *M1*)[*c1*]
   p2.**classes**(*c1*, *m2*, *M2*)[*c2*]
   p1.`inverse`[*p2*]
   For example, in Figure 1, the association of properties `compProd,compHardPart` does not have a name. It is defined via its properties, as follows:
   ```
   compProd.classes(ComputerHardware,0,*)[Computer];
   compHardPart.classes(Computer,1,1)[ComputerHardware];
   compProd.inverse(compHardPart);     % or vice versa
   ```

– *Class attribute declaration*: *CLASS*.**attr**(*ATTRIBUTE*)[*TYPE*]
  For example:
  ```
  ComputerProduct.attr(specification)[String];
  ```
– *Method declaration*:
  CLASS.`func`(*FUNC*(*PARAM* : *TYPE*, ...))[*RETURNTYPE*]
  For example:
  ```
  ComputerProduct.func(appendSpecification(input:String, passwd:int))[Boolean];
  ```
– *Class hierarchy declaration*: *SUBCLASS*::*SUPERCLASS*
  Here we use the PathLP subtyping notation. For example:
  ```
  ComputerSoftware::ComputerProduct;
  Computer::ComputerProduct;
  ComputerHardware::ComputerProduct;
  Application::ComputerSoftware;
  OperatingSystem::ComputerSoftware;
  Monitor::ComputerHardware;
  ```
– *Model constraints (invariants):* Model constraints are written using plain PathLP code. They can be captured using FOML constraints, or using inference rules. For example, the two constraints in Figure 1 are formulated as follows:
  1. *TechnicalSupport must support both ComputerHardware and ComputerSoftware of a Computer that it supports*:
     ```
     !- computer.TechnicalSupport[?ts],
        not computer.compSoftware.TechnicalSupport[?ts],
        not computer.compHardPart.TechnicalSupport[?ts];
     ```
  2. *A computer has at least 1 and at most 2 operating systems*:
     ```
     !- ?comp:Computer, bag(?os, (?comp.compSoftware[?os], ?os:OperatingSystem), ?list),
        ?list.length[?length], (?length < 1 or ?length > 2);
     ```

The overall code that describes the class model in Figure 1 is given in the Appendix.
You can see how FOML is loaded and how the diagrams are loaded in the file `runuser` (can be found in download package), which loads the example `tests/user` by calling

```
> pathlp runuser
```

2.1 New

A new syntax was added to FOML 1.3: an ability to express the model in regular PathLP syntax. For example, we can create a property `computer` from class `DesktopSoftware` to class `Desktop` by

```
DesktopSoftware!computer[Desktop];
```

and a link `os` from an object `dell` to an object `windows` by

```
dell.os[windows];
```

I created a small example in **PathLP/PathLP_public/foml1.3**. To run it, you need run from the **foml1.3** directory the command

> *> pathlp exstud*

This will load the files in the subdirectory **exstud**: **stud_metamodel.ppl**, **stud_model.ppl**, **stud_object.ppl**, with metamodel (almost empty file), model (new syntax) and object diagram accordingly. The result will be some constraint violations, because the new part of the code should be implemented soon. This part enriches the model by reasoning that if

```
c!p[c1];
c1!p1[c];
a1.props[c^c1];
o:c;
o1:c1;
o.p[o1];
```

meaning, `a1` is an association of two properties, `p` and `p1`, and `o` and `o1` are linked by instance of `p`, then `o1` and `o` are linked by an instance of `p1`:

```
o.p[o1];
```

After that, you can run the queries in **stud_queries_examples** and see the results.

3 Reasoning about FOML 1.2 Models

3.1 Model querying

Meta-level querying

Meta-level querying refers to queries about models. At the highest level, meta-meta-level refers to querying the elements of the meta-model. The FOML meta-meta model consists of a single meta-meta class, called `MM_Class`. Its elements can be retrieved by the following query:

```
?- ?x:MM_Class;
```

which binds ?x to each of the 8 meta-classes described in Section 1 in turn.

Querying model content

The syntactic elements of a model can be retrieved by querying the members of the meta classes

- `Class`
- `Association`
- `Property`
- `Attribute`
- `Multiplicity`
- `Generalization`

For example, for the class diagram in Figure 1, querying about the classes in this model looks like this:

```
?- ?x:Class;
?x = ComputerProduct;
?x = TechnicalSupport;
?x = ComputerSoftware;
?x = ComputerHardware;
?x = Computer;
?x = Application;
?x = OperatingSystem;
?x = Monitor;
```

Querying model structure:

- *Find source class of a property*: *PROP*.`source`[*CLASS*]. For example:

  ```
  ?- compProd.source(?c);
  ?c = Computer;
  ```

- *Target class of property*: *PROP*.`target`[*CLASS*]
- *Classes of a property*:
  PROP.`classes`(*SOURCECLASS*)[*TARGETCLASS*]
- *Class outgoing property*: *CLASS*.`prop`[*PROP*]
- *Property minimum multiplicity*: *PROP*.`min`[*MULTIPLICITY*]
- *Property minimum multiplicity*: *PROP*.`max`[*MULTIPLICITY*]
- *Shortened notation for a property connecting two classes*:
  SOURCECLASS.`property`(*PROP*)[*TARGETCLASS*]
- *Shortened notation for association with properties that connect classes*:
  ASSOC.`prop`(*PROP*)[*TARGETCLASS*]
- Association querying:
  1. Regular syntax:
     ASSOC.`prop`(*PROP1*, *LOW1*, *HIGH1*)[*TARGET1*]
     ASSOC.`prop`(*PROP2*, *LOW2*, *HIGH2*)[*TARGET2*]
  2. Property-wise declaration:
     PROP.`classes`(*SOURCE*, *LOW*, *HIGH*)[*TARGET*]
     PROP1.`inverse`[*PROP2*]
- *Direct subclass relation*: - not by transitivity
  SUBCLASS.`dirsub`[*SUPERCLASS*]
- Subclass querying:
  SUBCLASS::*SUPERCLASS*
- Class attribute querying: *CLASS*.`attr`(*ATTRIBUTE*)[*TYPE*]

Intensional queries of model structure – using operations on model elements:
FOML has a library of operations on classes and properties. These operations define *intensional* classes and operations, and enable rich analysis of model structure. The operations are: *compose, path, cycle, closure, property-or. property-not, class-or, class-and, class-not.* There is also *class-of* that generates an enumeration class.

- Path of properties that connect two classes:
  SOURCECLASS.`path`([*PATHLIST*])[*TARGETCLASS*]
  For example:

  ```
  ?- ComputerSoftware.path(?path)[?target];
  ?path = [computer,compHardPart],?target = ComputerHardware;
  ?path = [computer],?target = Computer;
  ```

- Cycle of properties involving a class:
  CLASS.`cycle`[*PATHLIST*]
  For example:

```
?- ?sourceclass.cycle(?model123)[?properties_cycle_list];
false
```

– Logical operations on classes and properties.
  - **OR**(*CLASS*1, *CLASS*2) - class disjunction. For example:
    ```
    ?- ?o:OR(OperatingSystem, TechnicalSupport);
    ?o = desktopOS;
    ```
  - **AND**(*CLASS*1, *CLASS*2) - class conjunction
  - **NOT**(*CLASS*) - class negation
  - **OR**(*PROP*1, *PROP*2) - property disjunction
  - **AND**(*PROP*1, *PROP*2) - property conjunction
  - **NOT**(*PROP*) - property negation

– Function querying:
  CLASS.func(*FUNCNAME*(*PARAMETER* : *TYPE_COMMA_SEPARATED_LIST*))[*RETURNTYPE*]

4 Specification of Instances (States) in FOML 1.2

This section shows the FOML specification for instances (states) of the supported class model, as described in Section 2. The specification is demonstrated using the object diagram in Figure 3, which is a legal instance of the class model in Figure 1.

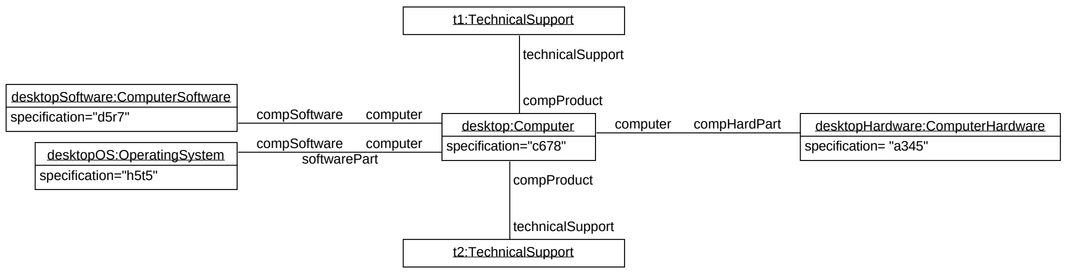

Fig. 3. An object diagram example

– *Object declaration*: *OBJECT*:*CLASS*
  The specification of the objects in Figure 3 is given below:

```
desktop:Computer;
desktopSoftware:ComputerSoftware;
desktopOS:OperatingSystem;
desktopHardware:ComputerHardware;
t1:TechnicalSupport;          % For example, t1 might be monitorSupport
t2:TechnicalSupport;          % and t2 might be cpuReplacement
```

– *Link declaration*: *C1.assoc_a*[*C2*]
  For example:

```
desktop.softwarePart[desktopSoftware];
desktop.softwarePart[desktopOS];
desktop.comPart[desktopHardware];
desktop.comTechSupport[t1];
desktop.comTechSupport[t2];
```

– *Object attribute declaration*:
  OBJECT.**attr**(*ATTRIBUTE*)[*VALUE*]
  For example:

```
desktop.attr(specification)[c678];
desktopSoftware.attr(specification)[d5r7];
desktopOS.attr(specification)[h5t5];
desktopHardware.attr(specification)[a345];
```

5 Reasoning about Model Instances (States)

– Membership querying: *OBJECT*:*CLASS*
  For example,

```
?- Desktop:?c;
?c = Computer;
?c = ComputerProduct
```

– Enumeration class: `classOf`(*MEMBERSLIST*) For example:

```
?- ?c:classOf(ComputerProduct, TechnicalSupport);
?c = ComputerProduct;
?c = TechnicalSupport
```

– *Direct membership relation*: - not by generalization. Meaning, the least possible class that an object belongs to in generalization sequence of classes.
  `object`(*OBJECT*, *MODNAME*).`dirmmb`[`class`(*CLASS*, *MODNAME*)] - direct membership
– Link querying:
  `object`(*SOURCE*, *MODNAME*).`link`(*PROP*)[`object`(*TARGET*, *MODNAME*)]
– Object attribute querying: `object`(*OBJECT*, *MODNAME*).`attr`(*ATTRIBUTE*)[*VALUE*]
  For example: Find all the attributes of objects in class ComputerSoftware.

```
?- ?o:class(ComputerSoftware, model123), ?o.attr(?attr)[?val];
?o = object(DesktopSoftware,model123),?attr = specification,?val = d5r7;
?o = object(DesktopOS,model123),?attr = specification,?val = h5t5;
```

References

1. Balaban, M., Kifer, M.: Logic-Based Model-Level Software Development with F-OML. In: MoDELS 2011. (2011)
2. Khitron, I., Kifer, M., Balaban, M.: PathLP: A Path-Oriented Logic Programming Language. The PathLP Web Site (2011) http://pathlp.sourceforge.net.
3. Kifer, M., Lausen, G., Wu, J.: Logical foundations of object-oriented and frame-based languages. Journal of ACM **42** (1995) 741–843
4. Khitron, I., Balaban, M., Kifer, M.: The FOML Site. `https://sourceforge.net/projects/pathlp/files/foml/` (2016)
5. Balaban, M., Maraee, A.: UML Class Diagram: Abstract syntax and Semantics. `http://www.cs.bgu.ac.il/~cd-patterns/?page_id=1695` (2014)

6 Appendix a

FOML 1.2 code of Figure 1:

```
%%% Classes
ComputerProduct:Class;
TechnicalSupport:Class;
ComputerSoftware:Class;
```

```
ComputerHardware:Class;
Computer:Class;
Application:Class;
OperatingSystem:Class;
Monitor:Class;
%%% Generalization
ComputerSoftware::ComputerProduct;
Computer::ComputerProduct;
ComputerHardware::ComputerProduct;
Application::ComputerSoftware;
OperatingSystem::ComputerSoftware;
Monitor::ComputerHardware;
%%% Associations
softwarePart.prop(computer,0,*)[Computer];
softwarePart.prop(compSoftware,1,1)[ComputerSoftware];

compProd.classes(ComputerHardware,0,*)[Computer];
compHardPart.classes(Computer,1,1)[ComputerHardware];
compProd.inverse[compHardPart];

comTechSupport.prop(compProduct,0,*)[ComputerProduct];
comTechSupport.prop(technicalSupport,0,*)[TechnicalSupport];

%%% Attributes
ComputerProduct.attr(specification)[String];
%%% Operations
ComputerProduct.func(appendSpecification(input:String,
        passwd:int))[Boolean];

%%% Membership
desktop:Computer;
desktopSoftware:ComputerSoftware;
desktopOS:OperatingSystem;
desktopHardware:ComputerHardware;
t1:TechnicalSupport;
t2:TechnicalSupport;
%%% Links
desktop.softwarePart[desktopSoftware];
desktop.softwarePart[desktopOS];
desktop.comPart[desktopHardware];
desktop.comTechSupport[t1];
desktop.comTechSupport[t2];
%%% Attributes
desktop.attr(specification)[c678];
desktopSoftware.attr(specification)[d5r7];
desktopOS.attr(specification)[h5t5];
desktopHardware.attr(specification)[a345];
```

7 Appendix b

FOML syntax:

```
<FOML-stmt> → <membership> | <subclass> | <property> | <inverse> | <link> | <classattr>
                  | <objattr> |<func> |<reasoned>
```

```
<membership> → <object> : <class>
<subclass> → <class> :: <class>
<property> → <assoc>.prop(<prop>, <min>, <max>)[<class>] |
<prop>.classes(<class>, <min>, <max>)[<class>]
<inverse> → <prop>.inverse[<prop>]
<link> → <object>.<prop>[<object>]
<classattr> → <class>.attr(<attr>)[<Type>]
<objattr> → <object>.attr(<attr>)[<value>]
<func> → <class>.func(<func>(<paramslist>))[<Type>]
<reasoned> → <prop>.source[<class>]
| <prop>.target[<class>]
| <class>.prop[<prop>]
| <prop>.min[<min>]
| <prop>.max[<max>]
| <class>.property[<class>]
| <assoc>.prop[<class>]
| <class>.dirsub[<class>]
| <class>.dirmmb[<class>]
| <class>.path(<proplist>)[<class>]
| <class>.cycle[<proplist>]
<class> → OR(<class>, <class>) | AND(<class>, <class>) | NOT(<class>) | <name>
<prop> → OR(<prop>, <prop>) | AND(<prop>, <prop>) | NOT(<prop>) | <name>
<assoc> → <name>
<attr> → <name>
<min> → <int>
<max> → ><int> | '*'
<paramslist> → '[' ((<param>, ) * <param>)? ']'
<proplist> → '[' ((<prop>, ) * <prop>)? ']'
<param> → <name> : <Type>
```