JOT
JOURNAL OF
OBJECT TECHNOLOGY

Logic-based Software Modeling with FOML

Mira Balaban[*], **Igal Khitron**[*], **and Michael Kifer**[†]
[*]Ben-Gurion University, Israel
[†]Stony Brook University, USA

ABSTRACT Models are at the heart of the emerging Model-Based Systems Engineering (*MBSE*) approach. MBSE is motivated by the growing complexity of software, which requires multiple levels of abstraction that programming languages do not support. In MBSE, models play a central role in the software evolution process. Rich model management must rely on a *unifying* underlying formal framework that can support, integrate, and mediate powerful modeling services. This paper describes FOML, a *F*ramework for *O*bject *M*odeling with *L*ogic, its realization in a modeling tool, proves the correctness of class modeling in FOML, illustrates the process of software modeling with the tool, and presents the main features of the system. The FOML framework for software modeling is compact yet powerful, formal, and is based on an underlying logic rule language called PathLP. The combination of class-based conceptualization with a formal logical base enables clean mediation and integration of a wide range of modeling activities and provides a provably correct formulation of class models. Our implementation of FOML features seamless integration of multiple modeling services that simultaneously support multiple models and provide reasoning, meta-reasoning, validation, testing, and evolution services.

KEYWORDS UML class diagrams, F-Logic, objects, constraints, types, model transformation, OCL, logic programming, model theory.

1. Introduction

Models are at the heart of the emerging *Model-Based Systems Engineering* (*MBSE*) approach. MBSE is motivated by the growing complexity of software, which requires multiple levels of abstraction that programming languages do not support (R. B. France et al. 2006; Schmidt 2006; R. France & Rumpe 2007). In MBSE, models play a central role in the software evolution process. Rich model management must rely on a *unifying* underlying formal framework that can support, integrate, and mediate powerful modeling services (Kleppe et al. 2003; Frankel 2003; Sendall & Kozaczynski 2003).

Intensive efforts in the modeling community in the last two decades have produced an impressive variety of tool support for models, and frameworks like Eclipse EMF, Epsilon (Kolovos et al. 2008), and MetaEdit, that supports *Domain Specific modeling* (Kern & Kuhne 2007). 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 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 FOML, a *F*ramework for *O*bject *M*odeling with *L*ogic,[1] its realization in a modeling tool, proves the correctness of class modeling in FOML, illustrates the process of software modeling with the tool, and presents the main features of the system (Khitron et al. 2011b). The FOML framework for software modeling is compact yet powerful and formal. It is based on an underlying logic rule language of *guarded path expressions*, called *PathLP*, which is used to define objects and their types.[2] The combination of class-based conceptualization and a formal logical base enables clean mediation and integration of a wide range of modeling activities and supports

JOT reference format:
Mira Balaban, Igal Khitron, and Michael Kifer. *Logic-based Software Modeling with FOML*. Journal of Object Technology. Vol. 19, No. 3, 2020.

http://dx.doi.org/10.5381/jot.2020.19.3.a19

[1] A preliminary version this work appeared in (Balaban & Kifer 2011).
[2] It is inspired primarily by F-logic (Kifer, Lausen, & Wu 1995), and also by HiLog (Chen et al. 1993) and Transaction Logic (Bonner & Kifer 1994).

a provably correct formulation of class models.

Our implementation of FOML features seamless integration of multiple modeling services that simultaneously support multiple models and provides reasoning, meta-reasoning, validation, testing, and evolution services.

This paper is organized as follows: Section 2 provides background on class and object modeling, together with a formal set-theoretic definition, Section 3 formally introduces the PathLP language. The FOML layer is described in Section 4, and the FOML tool is described in Section 5. Section 6 describes related work and Section 7 concludes the paper.

2. Background on UML Class and Object Models

Class models and object models provide static views of problem domains. They describe system structure in terms of classes, associations, and constraints. These models are the essence of the Unified Modeling Language (UML) (OMG 2017), a widely accepted standard for modeling software systems. UML consists of a variety of visual modeling diagrams, each describing a different aspect of software. *Class Diagrams*, the backbone of UML, are used to visualize class models; object diagrams visualize object models.

A class diagram consists of basic elements, descriptors, and constraints. The basic elements are *classes* and *associations*; the *descriptors* are class and association *attributes*, along with association *properties*; the constraints are restrictions imposed on these elements. The constraints include constrained elements—*association classes* and *aggregation/composition properties*, together with constraints on these elements: (1) *multiplicity* (or *cardinality*) constraints on properties and attributes, with or without *qualifiers*; (2) class hierarchy constraints; (3) generalization set constraints; and (4) inter association constraints.

Figure 1 shows a class diagram for modeling computer manufacturing software. It has the *classes* `Hardware, Software, GPU, Computer, OS, CompAppl, ComputingAPI` and `GPUAPI`, where `GPUAPI` is an *association class*. The *associations* are `hardwSoftw, partParent, gUnitComp, compOs, osCompAppl, alternative, compApplApi, gpuApi` and `CompApplGApi`. Each association has a pair of *properties* that are inverses of each other (also termed *roles, association-ends*), and each property has a *multiplicity* constraint. The class diagram also includes *class hierarchy* (*generalization/specialization*) constraints, a *generalization set* constraint, and three *subsetting* inter-property constraints. The diagram states (among other things) that instances of `Computer` must be related (via the association `compOS`) to one or more instances of class `OS`, which also happen to be instances of `Software`. Furthermore, `OS` instances must be disjoint from the instances of `CompAppl` due to the *disjoint* constraint that involves these classes. The association class `GPUAPI` is related to the `gpuApi` association (linking `GPU` and `ComputingAPI`), meaning that there is a 1:1 correspondence between instances of `GPUAPI` and links of `gpuApi`. This constraint is shown in the figure using a dashed line. The subsetting constraint between the *os* and the *softw* properties (and also between *compG* and *part*) states that *os* is a subproperty of *softw*, i.e., links of *os* are also links of *softw*.

The visual notation of class diagrams is said to be the *concrete syntax* of class models. The *abstract syntax* of class diagrams formally defines the semantically meaningful syntactical categories, and their inter-relationships. The semantics of class models is defined via the abstract syntax. We illustrate the abstract syntax and its semantics using the above example.

Object models describe data for class models, which includes objects, their content, and inter-relationships; they are visualized via object diagrams. An object model of a class model is assumed to be a *legal instance*, i.e., to satisfy the class diagram constraints. Based on this assumption, the object model can avoid explicit specification of derived information. Figure 2 presents an object diagram for a legal instance of the class model in Figure 1. Object memberships that are implied by class hierarchy (like `ThinkPad` being an object of `Hardware`) or links that are implied by subsetting (like `Linux` being a `softw` of `ThinkPad`), are not shown.

Class models can be extended with constraints, that are used to express intended relationships between objects in legal instance models of a class model. In UML, the standard constraint language is OCL (Object Management Group (OMG) 2012; Warmer & Kleppe 2003). It enables specification of invariants, queries, and pre/post conditions on operations. It is not a standalone language: its expressions are associated with UML diagrams.

For example, the class model in Figure 1 can be extended with the following constraint, written in OCL:

An application that runs with a GPUAPI with some GPU card must run on an operating system that runs on a Computer with that GPU unit:

```
Context CompAppl
inv: not self.applApi.apiGUnit->
            intersection(self.applOs.osComp.compG)->isEmpty()
```

In Section 4.1 we present this same constraint for the FOML encoding of Figure 1. In (Balaban et al. 2016), we compare class and object modeling using OCL, Alloy (Jackson 2002b) or FOML.

2.1. Class Models – Abstract Syntax

Class and *instance* models are defined over a global, sorted, infinite vocabulary $\mathcal{V} = \langle \mathcal{O}, \mathcal{C}, \mathcal{P}, \mathcal{A}, \mathcal{A}ttr, \mathcal{D}t \rangle$ of *object* ($\mathcal{O}$), *class* ($\mathcal{C}$), *property*[3] ($\mathcal{P}$), *association* ($\mathcal{A}$), *attribute* ($\mathcal{A}ttr$) and *datatype* ($\mathcal{D}t$) symbols; all these sets are disjoint.[4] For compactness, we omit elements like aggregation/composition, qualifiers and most inter-association/property constraints. For a full formalization of UML class models see (Balaban & Maraee 2017).

A *class model* over a vocabulary $\mathcal{V} = \langle \mathcal{O}, \mathcal{C}, \mathcal{P}, \mathcal{A}, \mathcal{A}ttr, \mathcal{D}t \rangle$ is a tuple $CM = \langle \mathcal{C}_{CM}, \mathcal{P}_{CM}, \mathcal{A}_{CM}, \mathcal{A}ttr_{CM}, \mathcal{D}t_{CM}, Mappings, Constraints \rangle$, where $\mathcal{C}_{CM} \subseteq \mathcal{C}$, $\mathcal{P}_{CM} \subseteq \mathcal{P}, \mathcal{A}_{CM} \subseteq \mathcal{A}$, $\mathcal{A}ttr_{CM} \subseteq \mathcal{A}ttr, \mathcal{D}t_{CM} \subseteq \mathcal{D}t$ are finite sets of Class, Property, Association, Attribute, and Datatype symbols that

[3] A property denotes a multi-valued function between classes. It corresponds to UML 2.5 binary-association end, with the descriptors isUnique=true, isOrdered=false.

[4] In multilevel modeling $\mathcal{O}$ and $\mathcal{C}$ can intersect (Balaban et al. 2018).

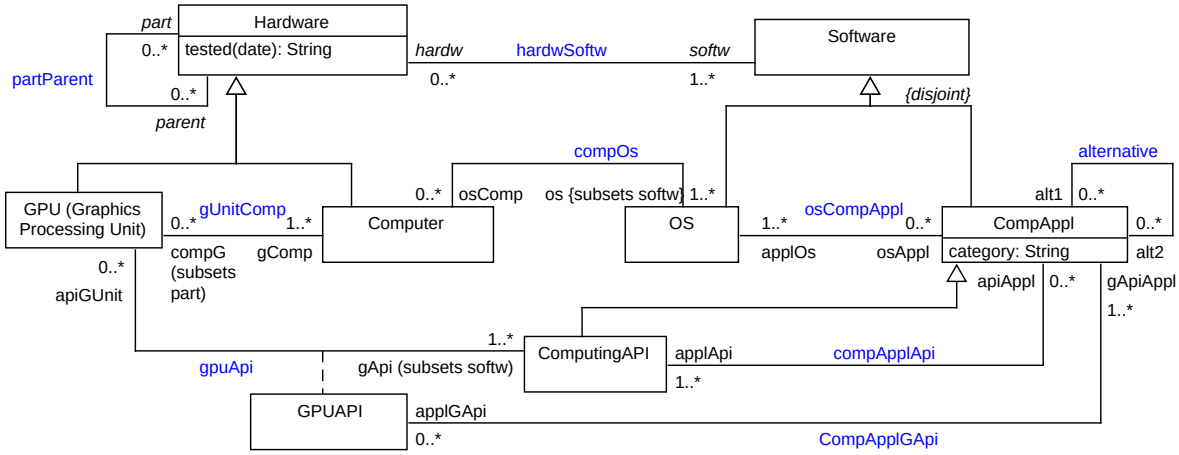

Figure 1 A class diagram for computer manufacturing

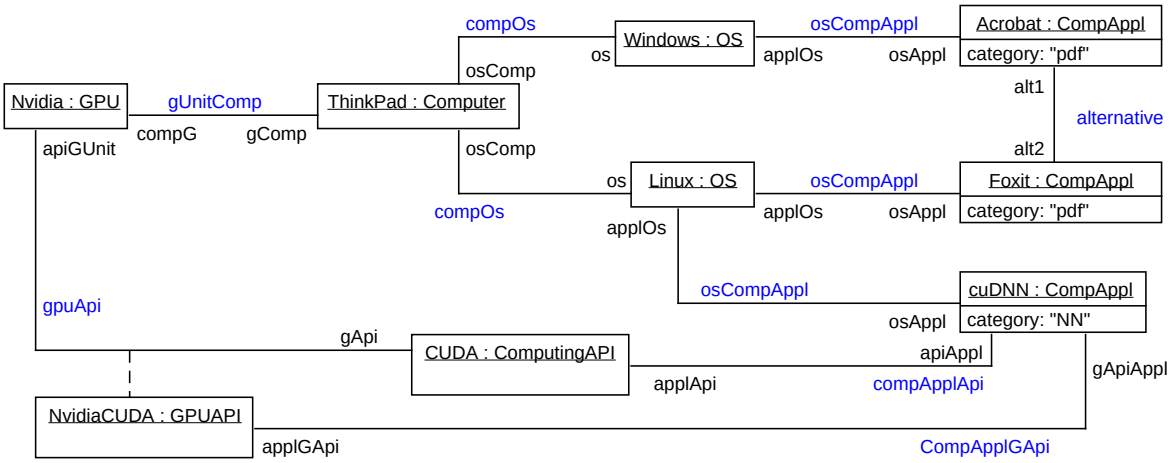

Figure 2 An object diagram for a legal instance of the computer manufacturing class model in Figure 1

actually appear in the class model. *Mappings* describe syntactic interrelationships between the symbols, and *Constraints* specify semantic requirements that can be imposed on the symbols. These sets are defined below.

Mappings:

- **Property mappings:**
  1. $inverse : \mathcal{P}_{\mathcal{CM}} \to \mathcal{P}_{\mathcal{CM}}$ is a bijective mapping, that assigns to *every* property p its unique inverse, denoted p^{-1}, such that $inverse(p) \neq p$, and $(p^{-1})^{-1} = p$.
  2. $source: \mathcal{P}_{\mathcal{CM}} \to \mathcal{C}_{\mathcal{CM}}$ and $target: \mathcal{P}_{\mathcal{CM}} \to \mathcal{C}_{\mathcal{CM}}$ are mappings such that $target(p) = source(p^{-1})$, which define the source and the target class of a property.

  In Figure 1, $softw = hardw^{-1}$, $target(softw) = source(hardw) = Software$ and $source(softw) = target(hardw) = Hardware$
- **Association mappings:**
  1. $props: \mathcal{A}_{\mathcal{CM}} \to \mathcal{P}_{\mathcal{CM}} \times \mathcal{P}_{\mathcal{CM}}$ is an injection such that $props(a) = \{p, p^{-1}\}$ and every property from $\mathcal{P}_{\mathcal{CM}}$ appears in exactly one $props(a)$.
  2. If p is a property, then $assoc(p)$ denotes the association such that $p \in props(assoc(p))$.
  3. For $a \in \mathcal{A}_{\mathcal{CM}}$, if $props(a) = \{p_1, p_2\}$ and $target(p_i) = C_i$, then we define $classes(a) = \{C_1, C_2\}$.

  For the model in Figure 1, we have:
  $props(hardwSoftw) = \{hardw, softw\}$,
  $assoc(softw) = assoc(hardw) = hardwSoftw$, and
  $classes(hardwSoftw) = \{Hardware, Software\}$.

Compact visual notation for associations: It is often convenient to use a compact notation that shows associations along with their properties, classes, and multiplicities. We write $a(C_1 \frac{p_1 \quad p_2}{m_1..M_1 \quad m_2..M_2} C_2)$ or $a(C_1 \frac{p_1 \quad p_2}{} C_2)$, if multiplicities are irrelevant) to denote an association a such that $props(a) = \{p_1, p_2\}$, $target(p_i) = C_i$, $min(p_i) = m_i$ and $max(p_i) = M_i$. For instance, the compact notation for association `hardwSoftw` in the schema of Figure 1, is $hardwSoftw(Hardware \frac{hardw \quad softw}{0..* \quad 1..*} Software)$.

– **Association class mappings:**

1. $\mathcal{AC} \subseteq \mathcal{C}_{\mathcal{CM}}$ is the subset of classes in C that function as *association classes*.
2. $assoc_{ac}\colon \mathcal{AC} \to \mathcal{A}_{\mathcal{CM}}$ is an injective function that maps association class symbols to association symbols. For example, In Figure 1, $assoc_{ac}(GPUAPI) = gpuApi$.

– **Attribute mappings**:

1. $att\colon \mathcal{C}_{\mathcal{CM}} \to \mathcal{A}tt_{CM}$ is a multivalued assignment of attribute symbols to classes.
2. For every class $C \in \mathcal{C}_{\mathcal{CM}}$, there is a partial mapping, $dt_C\colon att(C) \to \mathcal{D}t_{CM}$ that assigns datatypes to the attributes of C. Note that multiple classes can have the same attribute.

In Figure 1, $att(Hardware) = \{tested(date)\}$ and $dt_{Hardware}(tested(date)) = String$.

Constraints:

The class model constraints presented here are *property* and *attribute multiplicities, class hierarchy*, and *property subsetting*. We omit *aggregation/composition, generalization-sets*, and the inter-association/property constraints *redefinition, union, association-hierarchy, association-class hierarchy* and *xor*.

– **Multiplicity constraints – for properties and attributes**:

1. $min\colon \mathcal{P}_{\mathcal{CM}} \to \mathbb{N} \cup \{0\}$ and $max\colon \mathcal{P}_{\mathcal{CM}} \to \mathbb{N} \cup \{*\}$ assign minimum and maximum multiplicities to property symbols so that $min(p) \leq max(p)$ ($*$ denotes positive infinity).
2. For every class $C \in \mathcal{C}_{\mathcal{CM}}$, there are partial mappings, $min_C\colon att(C) \to \mathbb{N} \cup \{0\}$, and $max_C\colon att(C) \to \mathbb{N} \cup \{*\}$, as above.

– **Class hierarchy**: is an acyclic binary relation on class symbols in $\mathcal{C}_{CM}$: $C_2 \prec C_1$, means that C_2 is a subclass of C_1. The relation $\prec^+$ is a transitive closure of $\prec$ and $C_2 \preceq^* C_1$ stands for $C_2 = C_1$ or $C_2 \prec^+ C_1$. In Figure 1, class $Computer$ is a subclass of $Hardware$, i.e., $Computer \prec Hardware$.

– **Property subsetting (subproperties)**: is an acyclic binary relation $\prec$ on property symbols:[5] $p_1 \prec p_2$ says that p_1 *subsets* (is a *subproperty* of) p_2. As for classes, $\prec^+$ is the transitive closure of $\prec$ and $p_1 \preceq^* p_2$ stands for $p_1 = p_2$ or $p_1 \prec^+ p_2$.
The relation $p_1 \prec p_2$ satisfies these conditions: (i) $source(p_1) \prec^* source(p_2)$, (ii) $target(p_1) \prec^* target(p_2)$, and (iii) $max(p_1) \leq max(p_2)$.[6] In Figure 1, $sysSoft \prec softw$ means that if a `System` object `s` and a `Hardware` object `h` are linked by the `sysSoft` relation, then they are also linked by the `softw` relation.

[5] $\prec$ is overloaded for subproperties and subclasses.

2.2. Object Models (Instances) – Set-theoretic Semantics for Class Models

The standard set-theoretic semantics of class models associates such models with *instances* I, which consist of a semantic *domain* and a *denotation mapping* "$\cdot^I$" that assigns meaning to syntactic elements. Given a class model, class symbols are mapped to sets of objects in the domain, property symbols are mapped to multi-valued functions over these sets, and association symbols are mapped to relationships between these sets. The sets denoted by class and association symbols are called *extensions*. Attribute symbols are mapped to higher-order functions from class symbols to other functions from class extensions to data-type domains. For a symbol x, $\cdot(x)$, its denotation in I, is shortened into x^I.

Given a class model $CM = \langle \mathcal{C}_{CM}, \mathcal{P}_{CM}, \mathcal{A}_{CM}, \mathcal{A}ttr_{CM}, \mathcal{D}t_{CM}, Mappings, Constraints \rangle$. The formal specification for a restricted class model presented in this paper is defined as follows:[7]

Symbol denotations:

1. **Classes:** For $c \in \mathcal{C}_{CM}$, the extension of c in I, denoted c^I, is a set of elements in the semantic domain. The elements of class extensions are called *objects*.
2. **Properties:** For $p \in \mathcal{P}_{CM}$, its denotation is a multivalued function $p^I\colon source(p)^I \to target(p)^I$ such that $(p^{-1})^I = (p^I)^{-1}$.
3. **Associations:** For $a \in \mathcal{A}_{CM}$, a^I is a binary relationship over the extensions of the classes of a. If $props(a) = \{p, p^{-1}\}$ (enforced by the syntactic mappings), then the association denotes all object pairs that are related by its properties: $a^I = \{e \in source(p)^I,\ p^I(e)\}$. Elements of association extensions are called *links*.
4. **Datatypes:** Each data-type symbol T is associated with a known domain of values, denoted $domain(T)$.
5. **Attributes:** The denotation of an attribute is a higher-order function that maps a class symbol to a function from the class extension to the domain of the data-type of the attribute. That is, for $attr \in \mathcal{A}ttr_{CM}$, $attr^I$ is a partial function on $\mathcal{C}_{CM}$, such that for class $C \in \mathcal{C}_{CM}$, where attribute $attr \in att(C)$, $attr^I(C)\colon C^I \to domain(dt_C(attr))$ is a multi-valued function.

[6] Note that since the meaning of $\prec$ is subset of links, there is no restriction on minimum multiplicity.

[7] A full specification appears online in (Balaban & Maraee 2017).

Constraints:

1. **Multiplicity constraints on properties and attributes:**
   (a) **Properties:** For a property p, for every $e \in source(p)^I$, $min(p) \leq |p^I(e)| \leq max(p)$. The upper bound is ignored if $max(p) = *$.
   (b) **Attributes:** For a class C and attribute $attr \in att(C)$, for every object $e \in C^I$, $min_C(attr) \leq |attr^I(C)(e)| \leq max_C(attr)$. The upper bound is ignored if $max_C(attr) = *$.
2. **Association classes**: The association class constraint identifies the objects in the extension C^I of an association class C with the links in the extension of its associated association $assoc_{ac}(C)$. That is, there exists a 1:1 and onto semantic mapping $pairs_C$: $C^I \rightarrow (assoc_{ac}(C))^I$, that maps every object in C^I to a single link in the relation $(assoc_{ac}(C))^I$.
3. **Class-hierarchy constraints:** A constraint $C_1 \prec C_2$ denotes a subset relation between the class extension: ${C_1}^I \subseteq {C_2}^I$.
4. **Subsetting constraint:** For $p_1, p_2 \in \mathcal{P}$, $p_1 \prec p_2$ states that p_1 is a sub-mapping of p_2, i.e., for $e \in source(p_1)^I$, ${p_1}^I(e) \subseteq {p_2}^I(e)$.
   The semantics of subsetting requires the syntactic restrictions that for $p_1 \prec p_2$, the source and target classes of p_1 are descendant subclasses of the source and target classes of p_2, respectively, and the maximum multiplicity of p_1 can only restrict that of p_2. Moreover, in (Maraee & Balaban 2012) we show that the subsetting constraint is symmetric with respect to the inverse properties. That is, if $p_1 \prec p_2$ then also $p_1^{-1} \prec p_2^{-1}$.

Class model instances and semantic relationships: For a class model CM, its instances are denoted CM^I.

Objects and links: An *object* of CM^I is an element in the domain of I that belongs to the extension of some class. A *link* of CM^I is a pair of objects $\mathtt{o}_1, \mathtt{o}_2$ of CM^I, such that for some property p, $\mathtt{o}_2 \in \mathtt{p}^{\mathtt{I}}(\mathtt{o}_1)$. Links are visualized as $a(\mathtt{o}_1 \overset{p_1 \quad p_2}{\text{———}} \mathtt{o}_2)$, i.e., as labeled inverse edges between nodes o_1, o_2, where $props(a) = \{p_1, p_2\}$, and $\mathtt{o}_1 \in \mathtt{p}_1^{\mathtt{I}}(\mathtt{o}_2), \mathtt{o}_2 \in \mathtt{p}_2^{\mathtt{I}}(\mathtt{o}_1)$.

Instances: An instance I of a class model CM is **empty** if all its class extensions are empty; it is **non-empty** if all of its classes have non-empty extensions:[8] it is **finite** if all class extensions are finite; and it is **infinite** if some class extension is infinite. An instance I of a class model might or might not satisfy the constraints in the class model CM. If I satisfies all constraints in CM, denoted $I \models CM$, I is a *legal instance* of CM. I is a **partial instance** if it can be completed into a legal instance by addition of objects and links to class and association extensions. A class model is **satisfiable** if it has a legal instance, and is **finitely satisfiable** if it has a finite, non-empty legal instance (Berardi et al. 2005; Balaban & Maraee 2013). In software modeling we are interested in finitely satisfiable class models.

Compact instance representation: Instances of a class model can be represented as collections of their object memberships with their attributes, and of their links. This is the standard representation in object diagrams, as shown in Figure 2. However, object memberships and links that are *implied* by the class model semantics do not need to be specified. When some or all of the implied information is omitted in an instance, we call it a **compact instance specification**. For example, in Figure 2, the membership of `ThinkPad` in `Hardware`, which is implied from the class hierarchy constraint `Computer` $\prec$ `Hardware` in Figure 1, is not specified. Likewise, the subsetting constraint `os` $\prec$ `softw` (`os subsets softw`), in Figure 1, implies the missing link $\mathtt{HardwSoftw}(\mathtt{ThinkPad} \overset{\mathtt{hardw} \quad \mathtt{softw}}{\text{———}} \mathtt{Linux})$. On the other hand, the convention of object diagrams is that object memberships are always explicitly specified, although they are implied from the definition of associations in which objects are involved.

Herbrand instances: A **Herbrand instance**[9] of a class model CM over a global vocabulary $\mathcal{V} = \langle \mathcal{O}, \mathcal{C}, \mathcal{P}, \mathcal{A}, \mathcal{A}ttr, \mathcal{D}t \rangle$ is an instance of CM over the domain $\mathcal{O}$. Herbrand instances can be written using a set notation, that explicitly lists objects and links, i.e., $\{C_i = \{o_1^i, \ldots, o_{n_i}^i\}_{C_i \in \mathcal{C}_{CM}}, a_i = \{a(\mathtt{o}_1^{\mathtt{i}} \overset{p_1 \quad p_2}{\text{———}} \mathtt{u}_1^{\mathtt{i}}), \ \ldots, \mathtt{a}(\mathtt{o}_{\mathtt{n_i}}^{\mathtt{i}} \overset{p_1 \quad p_2}{\text{———}} \mathtt{u}_{\mathtt{n_i}}^{\mathtt{i}})\}_{\mathtt{a_i} \in \mathcal{A}_{\mathtt{CM}}}\}$. This writing saves explicit specification of property mappings and of empty extensions. The importance of Herbrand instances is in providing a convenient textual syntax for representing object diagrams.

Example 1. *A compact specification of the legal Herbrand instance H of the class model in Figure 1, that corresponds to the object diagram in Figure 2:*

$$
\begin{aligned}
H = \{&\mathtt{OS} = \{\mathtt{Linux}, \mathtt{Windows}\},\\
&\mathtt{CompAppl} = \{\mathtt{Foxit}, \mathtt{Acrobat}, \mathtt{cuDNN}\},\\
&\mathtt{Computer} = \{\mathtt{ThinkPad}\},\\
&\mathtt{GPU} = \{\mathtt{Nvidia}\},\\
&\mathtt{ComputingAPI} = \{\mathtt{CUDA}\},\\
&\mathtt{GPUAPI} = \{\mathtt{NvidiaCUDA}\},\\
&\mathtt{category}(\mathtt{Foxit}) = \text{"pdf"},\\
&\mathtt{category}(\mathtt{Acrobat}) = \text{"pdf"},\\
&\mathtt{category}(\mathtt{cuDNN}) = \text{"NN"}
\end{aligned}
$$

[8] Finer distinctions between instances with at least one class with non-empty extension exist, but they do not affect finite satisfiability and its complexity in UML class models (Balaban & Maraee 2013; Artale et al. 2010)

[9] By analogy with Herbrand interpretations in classical logic.

$$\texttt{osCompAppl} = \{\texttt{osCompAppl}(\texttt{Linux}\overset{applOs\quad osAppl}{\_\_\_\_\_}\texttt{Foxit}),$$
$$\texttt{osCompAppl}(\texttt{Windows}\overset{applOs\quad osAppl}{\_\_\_\_\_}\texttt{Acrobat}),$$
$$\texttt{osCompAppl}(\texttt{Linux}\overset{applOs\quad osAppl}{\_\_\_\_\_}\texttt{cuDNN})\},$$
$$\texttt{alternative} = \{\texttt{alternative}(\texttt{Acrobat}\overset{alt1\quad alt2}{\_\_\_\_\_}\texttt{Foxit})\},$$
$$\texttt{compOs} = \{\texttt{compOs}(\texttt{ThinkPad}\overset{osComp\quad os}{\_\_\_\_\_}\texttt{Linux}),$$
$$\texttt{compOs}(\texttt{ThinkPad}\overset{osComp\quad os}{\_\_\_\_\_}\texttt{Windows})\},$$
$$\texttt{gUnitComp} = \{\texttt{gUnitCompl}(\texttt{Nvidia}\overset{compG\quad Gcomp}{\_\_\_\_\_}\texttt{ThinkPad})\},$$
$$\texttt{gpuApi} = \{\texttt{gpuApi}(\texttt{Nvidia}\overset{apiGUnit\quad gApi}{\_\_\_\_\_}\texttt{CUDA})\},$$
$$\texttt{compApplGApi} =$$
$$\{\texttt{compApplGApi}(\texttt{cuDNN}\overset{gApiAppl\quad applGApi}{\_\_\_\_\_}\texttt{NvidiaCUDA})\},$$
$$\texttt{compApplApi} = \{$$
$$\{\texttt{compApplApi}(\texttt{cuDNN}\overset{apiAppl\quad applApi}{\_\_\_\_\_}\texttt{CUDA})\}\}$$

Besides objects and links, a legal instance of the class model in Figure 1 must specify the mapping $pairs_{\texttt{GPUAPI}}$ between objects of the association class GPUAPI and links of the association gpuApi. In the object model of Figure 2 this mapping is:
$pairs_{\texttt{GPUAPI}}(\texttt{NvidiaCUDA}) =$
$\texttt{gpuApi}(\texttt{Nvidia}\overset{apiGUnit\quad gApi}{\_\_\_\_\_}\texttt{CUDA})$ □

Finite instances I of CM over arbitrary semantic domains can be translated into corresponding Herbrand instances I_H, using a $1:1$ mapping I_H of the objects of CM^I to object symbols from the vocabulary $\mathcal{O}$. For a given I, I_H is obtained by replacing every object e in a class extension C^I by its symbol translation $I_H(e)$. The legal status of I with respect to CM can be checked by checking the legal status of I_H:

Claim 1. *For a finite instance I of CM and a corresponding Herbrand instance I_H, I is legal for CM if and only if I_H is legal.*

Proof. By building a correspondence between I_H and I, as in first-order logic. □

Semantic relationships: A class model constraint can be explicitly specified in the class model specification, or implied (derived) from other constraints. For example, transitivity of class hierarchy implies class hierarchy between a super class and all of its descendants; unspecified multiplicity constraints can derive from combinations of class hierarchy and property subsetting (Balaban & Maraee 2019); disjoint constraints can propagate from declared GS constraints (Balaban & Maraee 2013).

For a constraint γ, and an instance I, $I \models \gamma$ stands for "γ holds in I". If γ holds in every legal instance of a class model CM, we say that γ is *entailed* from CM, denoted $CM \models \gamma$. Clearly, all declared constraints are entailed from a class model. A class model CM_2 is *entailed* by a class model CM_1, denoted $CM_1 \models CM_2$, if every legal instance of CM_1 is a legal instance of CM_2. Class models are *equivalent*, denoted $CM_1 \equiv CM_2$, if they have the same set of legal instances.

Object-oriented characteristics of class modeling: The semantics of class model constraints satisfies the essential characteristics of object-oriented modeling:

1. *Transitivity of class hierarchy:* If $C_1 \prec C_2$ and $C_2 \prec C_3$, then $C_1 \prec C_3$.
2. *Supertype inheritance for properties and attributes:* For $p \in \mathcal{P}_{CM}$, if $target(p) = D$ and $D \prec^* SuperD$ then in every legal instance I of the class model, p^I has also type $SuperD$, i.e., for $e \in source(p)^I$, $p^I(e) \in SuperD^I$. The same holds for attributes of classes.
3. *Property and attribute inheritance for subtypes:* For $p \in \mathcal{P}_{CM}$, if $source(p) = C$ and $SubC \prec^* C$ then in every legal instance I of the class model, p^I is also defined on objects of $SubC$, i.e., for $e \in SubC^I$, $p^I(e) \in target(p)^I$. The same holds for attributes of classes.
4. *Polymorphic object typing, due to class hierarchy*: An object is multiply typed by all of its class ancestors. That is, in a legal instance I, $e \in C^I$ and $C \prec^* SuperC$ imply $e \in SuperC^I$. The same holds for attributes of classes.
5. *Object well typing*: A property (or attribute) is defined on all and only objects of its source class, and its values are objects (values) of its target class (type).

The five object-oriented characteristics of class modeling are used in Section 4 for showing the correctness of FOML encoding of class models and their instances.

3. PathLP – The Underlying Logic of FOML

This section describes the PathLP programming language (Balaban & Kifer 2011; Khitron et al. 2011a,b), an elegant logic programming language of *guarded path expressions*, inspired by F-logic (Kifer, Lausen, & Wu 1995). 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.

3.1. Syntax of PathLP

PathLP is a Logic Programming language, whose main feature is a construct called *path expression*. This construct describes object-attribute access, in the style of object-oriented paradigms. Following this intuition we use a node-edge metaphor in the presentation of PathLP. PathLP's path expressions generalize similar expressions in traditional imperative object-oriented languages. They extend a similar notion in XSQL (Kifer et al. 1992), an F-logic (Kifer, Lausen, & Wu 1995) based language designed for querying object-oriented databases, in the direction of the more general path expressions in the F-logic systems like FLORID and FLORA-2 (Frohn et al. 1998; Kifer 2007). PathLP expressions also have certain similarities with XPath (Deutsch et al. 1999).

3.1.1. Terms A PathLP term is a *constant*, a *variable*, or a *composite term*.

A **constant** symbol denotes an element in the domain of discourse, like a node, an edge, or a class in an object model. For example, `Person`, `child`, `John`, or `teach`. Constants are also known as 0-ary `functor symbols`, i.e., they take no arguments. The language of PathLP also has n-ary functors (or `function symbols`), $n > 0$, which take n arguments. They are used to define composite terms, as defined below.

A **variable** symbol is prefixed with a question mark `"?"`, e.g., `?accountNumber`, `?aPerson`. Variables get instantiated with concrete constants from a database in the course of query evaluation (cf. the `FROM`-variables in SQL). Sometimes the name of a variable is immaterial in which case we write just `"?"` and let the compiler invent a unique name.

A **composite term** specifies a tree-shaped data structure, like `mother_of(John)`, `semester(term(Spring),year(2020))`, or `course(Math, 1, 235, Mira)`. These terms are formally defined in Section 3.1.5.

3.1.2. Object and Type Path Expressions The main PathLP construct is a Path expression. Using the graph (node-edge) metaphor, path expressions represent queries over graphs. There are two kinds of path expressions: Object and Type. The building blocks of path expressions are *terms, guards, cardinalities*, and two *operators*: "." and "!". Intuitively, the "." operator provides navigation along *value paths*, while the operator "!" yields a *type* of an edge, rather than its value.

Object Path Expression: An **object path expression** is a basic formula in PathLP that selects a set of paths in an object-attribute graph. The general form of such an expression is

$\texttt{root.link}_1\texttt{[grd}_1\texttt{].link}_2\texttt{[grd}_2\texttt{]}\ldots\texttt{.link}_n\texttt{[grd}_n\texttt{]};$

where `root, link`$_i$ are terms that denote semantic entities, and `grd`$_i$ is a comma-separated list of such terms. The intuitive meaning is that `link`$_1$ applied at `root` evaluates to a set that contains all the terms in `grd`$_1$, `link`$_2$ applied at the result contains all the term listed in `grd`$_2$, etc. For example,

`John.teach[graphics,algorithms];`

if stated as a fact, says that both `graphics` and `algorithms` belong to the set of courses that `John` teaches. If stated as a query, the above asks if it is true that John teaches both of those courses. The target set and its size can be constrained by **type path expression** that are described below.

The general form of object path expressions allows for successive application of the "." operator, both in the base and in the guard parts. For example,

`John.teach[Mary.study, Jack.TA, ?h]`

is a query that asks whether John teaches courses that Mary studies, and also the courses in which Jack is a teaching assistant. It will also bind the variable `?h` to the courses taught by John. The expression

`John.study.teacher.age[?age],`

asks for the age of the teacher of the course that John studies. The expression

`a.b[?c].d.e[?f]`

asks for the middle and end nodes of paths going from node `a` through edges `b, d, e`.

An expression `a.b` (without a guard) denotes the set of end-nodes of edges `b` that start in `a`. The addition of a guard selects the end-nodes that are denoted by the guard. A path expression *without* a guard at the end can appear in a guard of another path expression, but it cannot be used as a fact or a query. An expression *with* a guard at the end can be used as a fact or a query, but not in a guard of another expression. An edge, node, or a guard in a path expression can be a variable. Intermediate (non-guard) terms that denote edges may or may not have guards. The first term cannot have a guard since it does not denote an edge.

Type Path Expression: This kind of path expression is intended to enforce types and size of node attribute values. They can also be used to query the type system. Type path expressions are similar to object path expressions except that they use "!" instead of ".": $\texttt{term!term}_1\texttt{[guard}_1\texttt{]!}\ \ldots\ \texttt{!term}_n\texttt{[guard}_n\texttt{];}$. For example,

`Lecturer!teach[Course];`

says (if stated as a fact) that if a lecturer (an object in class `Lecturer`) teaches something then that something must be an object in class `Course`. When posed as a query, the above asks if it is true that the type of the property `teach` in class `Lecturer` is declared to be `Course`. The syntax of type path expressions is similar to that of object path expressions with similar restrictions. A type path expression can have an optional guard, middle edges and internal guards. For example,

`Lecturer!teach[?course]!teaching_assistant[?TA]`

binds `?course` and `?TA` to the types declared for the properties `teach` and `teaching_assistant` in the appropriate classes.

Type path expressions introduce a new feature of a *multiplicity constraint*. A multiplicity constraint includes two natural numbers in non-decreasing order, and the last can also be `*` (infinity):

`term!term[guard]{multiplicity}`

For example,

`Lecturer!teach[Course]{3..4}`

states that every lecturer must teach at least 3 and at most 4 courses. A multiplicity constraint is optional, and its absence means unconstrained multiplicity.

3.1.3. Membership and Subtyping The PathLP language provides two special predicates, `":"` and `"::"`, to account for the class membership and subtyping relations, respectively. For example,

```
beatles:popgroup;
popgroup::musicgroup;
musicgroup::artgroup;
artgroup::artmaker;
artmaker:somethingmaker;
somethingmaker::somethingdoer;
```

The semantics of `"::"` and `":"` have the *transitivity of subtyping* and *transitivity of membership over subtyping* properties, so that the above implies:

`beatles:musicgroup`, `musicgroup:artmaker`, and `popgroup::artgroup`

PathLP is *partially typed* in the sense that not all typing

information must be explicitly specified. The language is polymorphic since a language construct might belong to multiple types.

3.1.4. Rules, Facts, and Queries The sentences of PathLP are the *facts, rules*, and *queries*, as usual in Logic Programming languages. In addition, PathLP has *constraints*.

A **fact** asserts an unconditioned snippet of knowledge. For example, these facts

```
a.b[c];
d:e;
```

assert that there is an edge labeled `b` from node `a` to node `c` and that object `d` is included in type `e`. Some more examples:

```
John.teach[chemistry];
John.teach[algebra];
```

is the same as the single fact using multiple guards:

```
John.teach[chemistry,algebra];
```

It says that there are two edges labeled `teach`; one connecting `John` to `chemistry` and another to `algebra`.

A **rule** specifies conditional knowledge. For example,

```
?c.e[f]:-?c:?d, ?d.e[f]
```

means that if an object `?c` is a member of a type `?d` that has an attribute `e` whose value is `f`, then `?c` also has attribute `e` with value `f`.

The head of a rule (the part left of `:-`) is an atomic formula and the body (the part right of `:-`) is a comma-separated sequence of atomic formulas, which is interpreted as a conjunction of those formulas. These atomic formulas can be including path expressions, membership, and subtyping assertions. The precise syntax of the rules is given in Section 3.1.5. Another example:

```
John.study[?course] :-
   Jack.study[?course], not Jorge.study[?course];
```

This states that John takes all courses that Jack takes except from those that Jorge takes also.

A **query** is a statement that starts with the symbol "`?-`" followed by a body—a conjunction of atomic formulas, which thus has the same syntax as a rule body. For example,

```
?- ?x:?c, ?c:d
```

asks a given set of facts and rules whether there is an instantiation for the variables `?x` and `?c` such that `?x` belongs to `?c`, and `?c` belongs to `d`. The results are returned as a set of tuples—each providing a requested instantiation. Another example:

```
?- ?person.study[chemistry, graphics];
```

The answer to this query is the set of all people (instantiations for `?person`) that study either chemistry or graphics.

Constraints are used to enforce semantic correctness in a domain and to reject illegal states. Constraints are formulated as formulas that characterize states that violate the intended semantics of a knowledge base. In PathLP, constraint checking is done on demand rather than in real time, but this is an implementation decision, not a semantic one. Constraints are distinguished by a special symbol "`!-`" followed by a body of the constraint. The latter has the same structure as the bodies of rules and queries. For example:

```
!- ?.b[?c], ?.d[?c]
```

specifies that an object `?c` cannot be the value of both attributes `b` and `d` of some objects; or using the graph view, a node `?c` cannot be the target of both edges `b` and `d`. The singleton symbol `?` denotes a "don't care variable." Each occurrence of such a symbol represents a new variable (whose name is immaterial).

Constraints are checked by presenting them as queries to a knowledge base of facts and rules. Since they specify forbidden situations, a constraint-query is expected to fail. Success of a constraint query means that the illegal situation is detected in the given knowledge base, and each answer to the query provides a witness for the violation. A more complex example:

```
!- ?person.study(?year1)[?course],
   ?person.teach(?year2)[?course],
   ?year2 < ?year1;
```

states that a person cannot teach a course before she studied it.

3.1.5. Formal 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, $n \geq 1$, are terms. The latter kind of a term is called a ***composite term***.

Path expressions: The following BNF productions define path expressions where the meta-symbols `Var`, `Term`, `NonNegativeInteger` denote variables, terms, and non-negative integers, respectively.

```
GuardedPE := GuardedObjPE | GuardedTypePE
UnguardedPE := UnguardedObjPE | UnguardedTypePE
GuardedObjPE := UnguardedObjPE  '[' Guard ']'
UnguardedObjPE := UnguardedExpr '.' (Expr '.')* UnguardedExpr
GuardedTypePE :=
       UnguardedTypePE '[' Guard ']' [ '{' Multiplicity '}' ]
UnguardedTypePE := UnguardedExpr '!' (Expr '!')* UnguardedExpr
Guard := Guard (',' Guard)* | UnguardedExpr | UnguardedPE
Expr := GuardedExpr | UnguardedExpr
GuardedExpr := UnguardedExpr '[' Guard ']'
UnguardedExpr := Term
Multiplicity := (NonNegativeInteger) '..' (NaturalNumber|'*')
```

Query formulas in PathLP are used as bodies of PathLP inference rules, queries, and constraints. They are defined as follows:

```
QueryFormula := ElementaryFormula
            | '(' QueryFormula ')' | 'not' QueryFormula
            | (QueryFormula ('and' | ',' | 'or') QueryFormula)
ElementaryFormula :=
                Membership | Subtype | GuardedPE | Comparison
Membership := Term ':'  Term
Subtype    := Term '::' Term
Comparison := Term Op Term
Op         := '=' | '!=' | '>' | '<' | '>=' | '=<'
```

Rules, queries, and constraints: Finally, we define PathLP ***rules***, ***facts***, ***constraints***, and ***queries*** via the following BNF:

```
Query       :=  '?-' QueryFormula ';'
```

```
Constraint  :=  '!-' QueryFormula ';'
Fact        :=  Consequent ';'
Rule        :=  Consequent ':-' QueryFormula ';'
```

A `Consequent` is an `ElementaryFormula` that can occur as a fact or rule head consequence. These are `ElementaryFormulas` that satisfy the following restrictions:

- They are not comparisons.
- Path expressions can have only one connective "." or "!" and only terms as guards (no path expressions in guards).

That is, only the following forms are allowed as rule heads or facts: `Term:Term`, `Term::Term`, `Term.Term[Term]`, `Term!Term[Term]`, or `Term!Term[Term]{Multiplicity}`.

3.2. Semantics

Universes. The *universe* U of PathLP includes a domain of entities D, over which various structures are defined: value graphs, type graphs, membership, inclusion relations, and multiplicity constraints.

The domain is uniform, and does not differentiate entities by their roles: `Node`, `edge`, or `type`. The same entity can play different roles depending on the syntactic context. For example, for an application dealing with university courses, D can include such entities as `Graphics`, `Algorithms`, `study`, `student`, `examine`, `teacher`, `John`, `Bradly`, `course`, `teach`, and so on. In the above, some entities are intended as attributes of other entities, and some might be types. For example, `study` can be an attribute of the student `John`, i.e., `John` might study `Algorithms`, and `teacher` can be the type of all individual teachers. At the same time, `teacher` can also be an attribute of courses, denoting the teachers of a given course. In sum, D is a set of entities that might play different roles in a variety of contexts. It is structured by the relations and functions that are defined on it, as described below.

A universe U includes a number of relations over its domain D: a binary relation $\in_D$, a partial order $\prec_D$ plus two ternary relations R_{val}, R_{type}. The relation $\in_D$ stands for membership, and the partial order $\prec_D$ is a weak version of subtyping. That is, $\texttt{a} \in_D \texttt{b}$ means that `a` is a member of `b`, when `b` plays the role of a type, and $\texttt{a} \prec_D \texttt{b}$ means that `a` is a subtype of `b`. In the university-courses domain, we can have

$Graphics \in_D cs\_course \prec_D course$,

$course \in_D interactive\_teaching\_tool \in_D educational\_framework$.

Note that this flexible structuring of membership and subtyping allows for multi-level domains, where types can be members of types in more abstract levels (Atkinson & Kühne 2001, 2008; Henderson-Sellers 2012).

As partial order, the subtype relation is transitive. In addition, the membership and subtype relations satisfy the *transitivity of membership over subtyping* constraint:

For any n, n', where $n \in_D n'$, if $n' \prec_D n''$ then $n \in_D n''$.

That is, the set of members of n' is a subset of the set of members of n''. Thus, the relations $\in_D$ and $\prec_D$ form a multi-level, intensional typed domain.

The ternary relation R_{val} represents the links that connect entities to properties and attributes, so one can view R_{val} as a directed graph over D. $R_{val}(n, e, v)$ means that the attribute `e` of the object `n` has the value `v`. For a given node n and edge e, there can be multiple such triples, i.e., the value graph allows multiple edges with the same label for a node. For example, in the university-courses domain, $R_{val}(John, study, Algorithms)$ means that the value of the attribute `study` of `John` *includes* `Algorithms`.

The ternary relation R_{type} specifies the types of entity attributes. $R_{type}(n, e, v)$ means that for every entity `n'` $\in_D n$, the values of the attribute `e` of n' are members of the type `v`. That is, for every $n' \in_D n$, $R_{val}(n, e, v')$ implies $v' \in_D v$. For example, in the university-courses domain, if $John \in_D student$, and $R_{type}(student, study, cs\_course)$ then $\texttt{Algorithms} \in_D \texttt{cs\_course}$.

Closure properties of R_{type} with respect to the subtype relation:

- *Upward-closure*: If attribute `e` of `n` has type `t`, then every supertype t' of `t` is also a type of `e` on `n`. That is, If $(n, e, t) \in R_{type}$ and $t \prec_D t'$ then also $(n, e, t') \in R_{type}$.
- *Inheritance*: If attribute `e` of `n` has type `t` then it has type t for every subtype n' of n, i.e., `e` is inherited. In other words, If $n' \prec_D n$ and $(n, e, t) \in R_{type}$ then $(n', e, t) \in R_{type}$.

The typing of an entity-attribute can be strengthened to include attribute size, i.e., the cardinality of attribute values. This is achieved with partial functions $D_{min} : D \times D \longrightarrow Integer$ and $D_{max} : D \times D \longrightarrow (Integer \cup \{*\})$, which provide constraints on the size of attribute values. Given a pair of entities n and e, these functions are either both defined or both undefined and they satisfy the constraint $0 \leq D_{min}(n, e) \leq D_{max}(n, e)$ (where $i < *$ for any integer i). The size restrictions are imposed by the requirement of *well-typing* for universes, as described below.

Well-typed universes. So far, R_{val} and R_{type} have not been related to each other. There can be triplets $(n, e, v) \in R_{val}$ for which no type restriction exists, i.e., there may be no n' such that $n \in_D n'$ and $(n', e, v) \notin R_{type}$. The well-typing constraint, which was first introduced in (Kifer, Lausen, & Wu 1995), characterizes universes in which all attribute values are typed. Well typing has two aspects: typing restriction on attribute values and multiplicity restrictions.

A universe U is ***well-typed*** if

- *Full typing:* For every value-triple $(n, e, v) \in R_{val}$, there is a type-triple $(n', e, t) \in R_{type}$, such that $n \in_D n'$ is satisfied.
- *Type inheritance:* For every value and type triples $(n, e, v) \in R_{val}$, $(n', e, t) \in R_{type}$, if $n \in_D n'$ then $v \in_D t$.
- *Multiplicity restriction:* For every n' and e for which $D_{min}(n', e)$, $D_{max}(n', e)$ are both defined, and for ev-

ery $n \in_D n'$, the number of edges going out of e is at least $D_{min}(n', e)$ and at most $D_{max}(n', e)$. That is, $D_{min}(n', e) \leq |\{v \mid R_{val}(n, e, v)\}| \leq D_{max}(n', e)$.

If the first restriction above is omitted, the universe U is said to be ***partially*** well-typed.

For example, in the university-courses domain, $D_{min}(student, study) = 2$ and $D_{max}(student, study) = 6$, means that a student can study between 2 to 6 courses a semester.

In addition, there is a mapping $F_D : D \longrightarrow (\times_{n=1}^{\infty} D \longrightarrow D)$ that associates every entity in D with a variadic function on D. This is used to interpret the functors in the language of PathLP as functions over D. Variadic functions are used here because functors with variable numbers of arguments is a common feature in Logic Programming languages, and it was found beneficial in PathLP as well. For example, in the university-courses domain, the entity `study` might be associated with a variadic function that maps `Spring2020` to the attribute "courses of semester Spring 2020" of students, so there might be a triple like this: $R_{val}(John, F_D(study)(Spring2020), Algorithms)$. In terms of the directed graph view of the PathLP domain of discourse, we interpret $F_D(study)(Spring2020)$ as an element of D, and in the last example, as an attribute of `John`.

Summary: A universe U is a tuple $\{D, \in_D, \prec_D, R_{val}, R_{type}, D_{min}, D_{max}, F_D\}$. The membership and subtyping relations partially simulate the properties of the membership and subset relations of set theory. They represent intensional but not extensional set relations. This means that PathLP sets that have exactly the same members are not necessarily the same sets, which is common in object-oriented languages. Likewise, if all members of type `t` are also members of t' then it still is not guaranteed that $t \prec_D t'$.

Interpretations

A PathLP ***interpretation***, $\mathcal{I}$, is a triple of the form $\langle U, I_C, I_V \rangle$, where $U = \{D, \in_D, \prec_D, R_{val}, R_{type}, D_{min}, D_{max}, F_D\}$ is a well-typed universe, as described earlier in this section, I_C is a mapping for constant symbols in PathLP, i.e., $I_C : Constant \longrightarrow D$, and I_V is a variable assignment for variable symbols of PathLP, i.e., $I_V : Var \longrightarrow D$.

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, since the context disambiguates the uses. The definitions of the denotation mapping and of satisfaction of formulas by interpretation 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 composite term $c(t_1, ..., t_n)$ (an unguarded expression) with zero or more arguments then:
  $\mathcal{I}(\tau) = \{I_{F_D}(I_C(c))(t'_1, ..., t'_n)\}$, where $t'_i \in \mathcal{I}(t_i)$, for $i = 1, ..., n$.

The above 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.

- *Object path expression:*
  - *Unguarded object path expression*: if τ has the form $objpathexp\,.\,expr$, where *objpathexp* is an object path expression and *expr* is a term then:

$$\mathcal{I}(\tau) = \{v \mid \exists n \in \mathcal{I}(objpathexp),\ \exists e \in \mathcal{I}(expr),\ \text{where } (n, e, v) \in R_{val}\}.$$

    That is, $obj\,.\,expr_1\,.\,expr_2\,.\ \ldots\ .\,expr_n$ denotes the set of nodes reachable from node $\mathcal{I}(obj)$ by a path labeled $\mathcal{I}(expr_1), \mathcal{I}(expr_2), \ldots, \mathcal{I}(expr_n)$.
    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:

$$\mathcal{I}(\tau) = \mathcal{I}(ungobjpathexp) \cap \mathcal{I}(grd, ungobjpathexp)$$

    where

$$\mathcal{I}(grd, ungobjpathexp) =$$
$$if \quad \text{for each } i = 1, ..., n,\ \mathcal{I}(ungobjpathexp) \cap \mathcal{I}(ungpathexp_i) \neq \emptyset$$
$$\text{then } \mathcal{I}(ungpathexp_1) \cup \ldots \cup \mathcal{I}(ungpathexp_n)$$
$$\text{else } \emptyset$$

    This definition ensures that $obj.pathexp[val_1, val_2]$ holds if and only if $obj.pathexp[val_1]$ and $obj.pathexp[val_2]$ both hold.
- *Type path expression*:
  - *Unguarded type path expression:* If τ is $tpathexp\,!\,expr$, where *tpathexp* is a type path expression and *expr* is an expression, then:

$$\mathcal{I}(\tau) = \{v \mid \exists n \in \mathcal{I}(tpathexp),\ \exists e \in \mathcal{I}(expr),\ \text{such that } (n, e, v) \in R_{type}\}.$$

  - *Guarded type path expression*: Similar to guarded object path expressions, but with a multiplicity constraint (the default is $\{0..*\}$):
    If τ is $ungtpathexp[grd]\{lo..hi\}$, where *ungtpathexp* is an unguarded type path expression, *expr*

is an expression, and grd is a guard of the form grd_1, ..., grd_n, then:

$$\mathcal{I}(\tau) = \mathcal{I}(ungtpathexp) \cap \mathcal{I}(grd, ungtpathexp)$$

where

$$\begin{array}{l}\mathcal{I}(grd, ungtpathexp) = \\ \quad if \quad \text{for each } i = 1, ..., n, \\ \qquad \mathcal{I}(ungtpathexp) \cap \mathcal{I}(grd_i) \neq \emptyset, \\ \qquad card_{min}(ungtpathexp) \geq lo, \text{and} \\ \qquad card_{max}(ungtpathexp) \leq hi \\ \quad \text{then } \mathcal{I}(grd_1) \cup \ldots \cup \mathcal{I}(grd_n) \\ \quad \text{else } \emptyset\end{array}$$

Here $card_{min}$ and $card_{max}$ are defined as follows (where $ungtpathexp = tpexp!expr$):

$$\begin{array}{l}card_{min}(tpexp!expr) = \\ \quad min\{D_{min}(n, e) \mid n \in \mathcal{I}(tpexp), e \in \mathcal{I}(expr)\} \\ card_{max}(tpexp!expr) = \\ \quad max\{D_{max}(n, e) \mid n \in \mathcal{I}(tpexp), e \in \mathcal{I}(expr)\}\end{array}$$

Satisfaction by interpretations:

We now define the logical satisfaction relation $\mathcal{I} \models \phi$ between PathLP interpretations $\mathcal{I}$ and formulas ϕ recursively as follows:

1. *Elementary formulas:*
   - *Membership*: $\mathcal{I} \models t : s$, where t, s are terms, if and only if $\mathcal{I}(t) \in_D \mathcal{I}(s)$.
   - *Subtyping*: $\mathcal{I} \models t :: s$, where t, s are terms, if and only if $\mathcal{I}(t) \prec_D \mathcal{I}(s)$.
   - *Guarded path expression with or without a multiplicity constraint*: $\mathcal{I} \models p$, 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, if and only if $\mathcal{I}(t) = \mathcal{I}(s)$. Likewise, $\mathcal{I} \models t < s$, if and only if $\mathcal{I}(t) < \mathcal{I}(s)$ (assuming $\mathcal{I}(t)$, $\mathcal{I}(s)$ evaluate to numbers). The definition of satisfaction for the remaining comparisons is similar.
2. *Query formulas:*
   - *And*: $\mathcal{I} \models t \text{ and } s$ if and only if $\mathcal{I} \models t$ and $\mathcal{I} \models s$.
   - *Or*: $\mathcal{I} \models t \text{ or } s$ if and only if either $\mathcal{I} \models t$ or $\mathcal{I} \models s$.
   - *Not*: $\mathcal{I} \models not\ t$ if and only if 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 ***legal interpretation*** (or a "*model*" in the logic terminology) of that specification. As usual in logic programming, we focus on ***canonical legal interpretations***. Without negation (`not`) and constraints, there is a unique least interpretation (Lloyd 1987), which is taken as the canonical interpretation. With negation (but ignoring the constraints), canonical interpretations are defined as three-valued *well-founded* interpretations (Van Gelder et al. 1991), which generalizes the concept of a least interpretation. Any PathLP specification (leaving aside the constraints) has a unique well-founded interpretation. We will not define such interpretations here because this is quite involved and is not needed for understanding the rest of the paper. If a canonical interpretation satisfies the constraints of a PathLP interpretation then it is also a legal canonical interpretation. Note that even though a canonical interpretation always exists and is unique, it may not satisfy the constraints and thus no *legal* canonical interpretation may exists.

A PathLP specification is ***satisfiable*** if it has a canonical legal interpretation. An *answer* to a query `?-` *queryformula* is the set of all instantiations of variables in *queryformula*, such that it is satisfied in the canonical legal interpretation.

Without negation, PathLP reduces to classical logic analogously to the reduction of F-logic to classical logic (Kifer, Lausen, & Wu 1995), and it is semi-decidable. With negation, it reduces to logic programs with the *well-founded semantics* (Van Gelder et al. 1991) and can be implemented on top of a tabling deductive engine, like XSB (Swift & Warren 2011), similarly to the Flora-2 implementation of F-logic (Kifer 2007; Yang & Kifer 2003). Without function symbols, PathLP is decidable and has polynomial data complexity, even with negation.

Object-oriented characteristics of PathLP: The semantics of PathLP satisfies the essential characteristics of object-oriented modeling (note the analogy with the object-oriented characterization of class modeling, at the end of Section 2.2):

1. transitivity of subtyping:
   $?Sub ::?C \ :- \ ?Sub ::?MidC, \ ?MidC ::?C;$
2. *inheritance of supertypes by properties and attributes*:
   $?C!?prop[?SuperT] \ :- \ ?C!?prop[?T], ?T ::?SuperT;$
3. *property/attribute inheritance by subtypes*:
   $?SubC!?prop[?T] \ :- \ ?C!?prop[?T], ?SubC ::?C;$
4. type membership through subtyping:
   $?obj :?C \ :- \ ?obj :?MidC, \ ?MidC ::?C;$
5. the *well-typing* constraints from Section 3.2:
   $!- \ ?obj.?prop[?val], \ not\ (?C!?p[?T], ?obj :?C);$
   $!- \ ?C!?prop[?Type], \ ?obj :?C, \ ?obj.?prop[?val], \ not\ ?val :?Type;$

4. FOML – A Language for Class and Object Modeling

FOML is a conceptual layer on top of PathLP that is intended to support object modeling. It is built to directly represent class and object models, and to support metamodeling. It can represent multilevel and domain specific modeling (Balaban et al. 2018) and can also support multiple conceptual models. This section describes the FOML language, its capabilities as a modeling language, and proves the correctness of its class and object modeling.

FOML (Khitron et al. 2017) naturally supports model-level activities, such as constraints and inference rules, extending explicit class modeling with UML diagrams, dynamic compositional modeling (intensional and transformational), reasoning about models (e.g., on-the-fly querying), model testing, meta-reasoning, which is used for analysis of models, and meta-modeling which can be used for Domain Specific Modeling. Meta-modeling in FOML relies on the uniform status of types and instances in PathLP, and it is being used for multilevel modeling. As an executable modeling language, FOML can express and reason about multiple crosscutting multilevel dimensions.

As a modeling language, FOML can support both model and metalevel modeling. At the model level, FOML can account for modeling diagrams, reasoning extensions, constraints, and query-answering. At the meta-level, FOML can be used for model analysis, for reasoning about model properties, and for checking structure and inter-relationships of models.

4.1. Model Level Modeling with FOML

FOML provides a textual encoding for class and object models, using PathLP statements, mainly type and object path expressions, and subtyping and membership facts.

Class model encoding in FOML: A class model consists of declarations of classes, their attributes, properties and associations, plus optional class-model constraints. PathLP type path expressions are used to specify classes with their properties and attributes, and the associated multiplicities. PathLP subtyping is used for class hierarchy constraints. All other class model constraints, and declaration of associations and their properties are specified using PathLP object path expressions, together with FOML reserved keywords, which are marked with the "$" prefix. The FOML account for the additional constraints is formulated within PathLP. Section 4.2.2 defines the three class model constraints in Figure 1: *generalization-set, association class* and *subsetting*.

As a reasoning system, FOML can infer derived status of elements. Therefore, derived elements are not explicitly declared in the encoding. In particular, inverse properties can be inferred from association declaration, default multiplicities ($\{0..*\}$) and default types ($\$Any$) can be inferred and omitted, class hierarchy can be inferred from GS (generalization-set) constraints, and multiplicity can be inferred from subsetting constraints. The FOML encoding of the class model in Figure 1 is shown in Listing 1. In the meta modeling section below, we show how FOML accounts for the intended meaning of the builtin constraints.

```
GPU::Hardware;
Computer::Hardware;
% generalization set constraint
Software.$GS(OS,CompAppl)[disjoint];
ComputingAPI::CompAppl;

Hardware!tested(date)[$String];
Hardware!part[Hardware];
partParent.$assocProperty[part,parent];
Hardware!softw[Software]{1..*};
hardwSoftw.$assocProperty[hardw,softw];

GPU!gComp[Computer]{1..*};
gUnitComp.$assocProperty[gComp,compG];
% subsetting constraint
compG.$subsets[part];

GPU!gApi[ComputingAPI]{1..*};
gpuApi.$assocProperty[gApi,apiGUnit];
% subsetting constraint
gApi.$subsets[softw];
% association class constraint
gpuApi.$assocClass[GPUAPI];

Computer!os[OS]{1..*};
compOs.$assocProperty[os,osComp];
% subsetting constraint
os.$subsets[softw];

CompAppl!applOs[OS]{1..*};
osCompAppl.$assocProperty[osAppl,applOs];
CompAppl!alt1[CompAppl];
alternative.$assocProperty[alt1,alt2];
CompAppl!category[$String];

CompAppl!applApi[ComputingAPI]{1..*};
compApplApi.$assocProperty[apiAppl,applApi];
GPUAPI!applGApi[CompAppl]{1..*};
compApplGApi.$assocProperty[gApiAppl,applGApi];
```

Listing 1 Example of FOML encoding of the class model of Figure 2

Object model encoding in FOML: An object model (instance) for a class model CM consists of object memberships of classes of CM, their attribute values, and their links. In Herbrand instances, the objects are symbols from $\mathcal{O}$. FOML encodes finite Herbrand object models, using: (1) PathLP membership facts for object membership encodings, and (2) PathLP object path expressions for link and attribute encodings. Like the compact representation of object models, the FOML encoding is compact, i.e., relying on the inference capabilities of FOML, it avoids declaration of derived object memberships and links. For example, inverse links are not declared, object memberships are declared only if they are not part of any link. Moreover, object memberships and links are inferred from class model constraints like class hierarchy and subsetting, and also based on user inference rules that are associated with the class model.

A compact FOML encoding of the object model from Figure 2 is shown in Listing 2. The object-level mapping of an association class constraint, enables the direct path between the `NvidiaCUDA` object of the association class `GPUAPI` and its associated objects `CUDA, Nvidia`.

```
ThinkPad.os[Linux,Windows];
ThinkPad.compG[Nvidia];

Linux.osAppl[Foxit];
Windows.osAppl[Acrobat];

Acrobat.category["pdf"];
Acrobat.alt2[Foxit];

NvidiaCUDA:GPUAPI;
NvidiaCUDA.gApi[CUDA];       % association-class
NvidiaCUDA.apiGUnit[Nvidia]; % links

cuDNN.applOs[Linux];
cuDNN.category["NN"];
cuDNN.applGApi[NvidiaCUDA];
```

Listing 2 Example of FOML encoding of the object model of Figure 2

This compact encoding leaves out multiple derived object memberships and links. For example, the link `Nvidia.gApi[CUDA]` is implied from the `$assocClass` mapping of `NvidiaCUDA` to the link `(Nvidia,CUDA)` of association `gpuApi`, and association class rule (4) in Section 4.2.2 (page 15). The link `ThinkPad.softw[Linux]` is implied from the `subsetting` constraint on `os`. Class memberships of objects derive from property specification in the class model, and from class hierarchies.

Two links in the compact encoding, are missing, since they are derived from user rules, in Subsection 4.1.1 (page 14): `cuDNN.applApi[CUDA]` is implied from user inference rule (2) on page 14, and `Foxit.category["pdf"]` is implied from the user inference rule that infers a common category for alternative computer applications, also on page 14.

Correctness of FOML modeling of class and object models: We show that for *basic class models*, that include only classes, properties, and multiplicity and class hierarchy constraints, the FOML encoding preserves the legal status of an instance. Extension to include attributes can be similarly proved. Extension for additional class model constraints depend on their FOML axiomatization (see Section 4.2.2) and is beyond the scope of this paper.

Definition 1 (FOML encoding for basic class models and Herbrand instances)**.** The FOML encodings of a class model CM and of a finite Herbrand instance H, are denoted CM^{FOML} and H^{FOML}, respectively.
Let CM be a basic class model and H be its valid Herbrand instance.

– *Construction of* CM^{FOML}*:*
  - *If* CM *includes an association* $a(C \frac{q \quad p}{n..N \quad m..M} D)$ *then* CM^{FOML} *includes type path expressions* `C!p[D]{m..M}`, `D!q[C]{n..N}`, *and the rules*
    $?o.p[?u] :- ?u.q[?o];$
    $?o.q[?u] :- ?u.p[?o];$
  - *If* CM *includes a class hierarchy constraint* $C \prec D$, *then* CM^{FOML} *includes the subtyping fact* `C :: D`,

– *Construction of* H^{FOML}*:*
  - *for any object symbol* o *such that* $o \in C^H$, H^{FOML} *includes the membership fact* $o : C$.
  - *if a link* $a(\mathtt{o} \frac{p \quad q}{} \mathtt{u})$ *is in* H *then* H^{FOML} *includes the object path expression facts:* `o.q[u]` *and* `u.p[o]`.

As an application of Logic Programming, the semantics of FOML is determined with respect to the ***canonical interpretation*** of $H^{FOML} \cup CM^{FOML}$. In general, the definition of canonical interpretation is quite involved (FOML uses Herbrand *well-founded interpretations* (Van Gelder et al. 1991)), but for basic class models the canonical interpretation is simply the *least Herbrand interpretation* that satisfies $H^{FOML} \cup CM^{FOML}$ (Lloyd 1987). A canonical interpretation always exists, is unique, and has a number of convenient properties.

We say that H^{FOML} is a ***valid FOML instance*** for CM^{FOML} if $H^{FOML} \cup CM^{FOML}$ has a well-typed (see Section 3.2) canonical interpretation. Note that even though a canonical interpretation always exists, a *well-typed* canonical interpretation may not. The following claim states that the encoding of *basic* class models is correct with respect to finite instances.

Claim 2 (Correctness of FOML encoding for basic class models)**.**
Let CM *be a basic class model and* H *be a finite Herbrand instance of* CM*. Then* H^{FOML} *is a valid FOML instance of* CM^{FOML} *if and only if* $H \models CM$.

Proof. (Sketch) CM^{FOML} and H^{FOML} are constructed by the rules shown in Definition 1. This construction guarantees the following:

1. property typing, their multiplicity constraints and class hierarchy constraints stand in 1:1 correspondence with type path expressions and subtyping facts in CM^{FOML}
2. object memberships and links in H stand in 1:1 correspondence with object path expressions and membership facts in H^{FOML}.

Based on the construction of the FOML encoding, and the common object-oriented characteristics of legal PathLP interpretations (listed at the end of Section 3) and of class modeling, it can be shown that H satisfies the constraints in CM if and only if $H^{FOML} \cup CM^{FOML}$ has a well typed canonical interpretation. □

An important aspect of FOML involves querying and inference. Since the semantics of FOML is defined through the canonical interpretation of $H^{FOML} \cup CM^{FOML}$, query answering reduces to the evaluation of queries in that canonical interpretation. It can also be shown that for a variable-free FOML query γ, if it holds in the canonical interpretation of $H^{FOML} \cup CM^{FOML}$ then γ holds in H.

4.1.1. Querying, constraining and extending class models.

A modeler can query a given object model of a class model and can extend a declared class model using inference rules written in PathLP under FOML conventions. Such rules

can infer new data elements for an object model, like new links between objects, infer missing attribute values, or derive class memberships. The rules can also define new *intensional elements*, i.e., model elements that are constructed based on explicitly declared ones.

Query-answering. Find GPU units and their computers:

```
?- ?gpu.gComp[?c], ?c:Computer;
```

Answer:

```
?gpu = Nvidia,
?c = ThinkPad.
```

Find all Software objects that have a related hardware:

```
?- ?soft.hardw[?hard];
```

Answers:

```
?soft=Linux,            %% Answer 1
?hard=ThinkPad;
?soft=Windows,          %% Answer 2
?hard=ThinkPad;
?soft=CUDA,             %% Answer 3
?hard=Nvidia.
```

The first query is answered, based on the inverse property and property declaration in the class model specification. The second query is answered based on the semantics of the *subsetting* constraint.

Constraining Class Models: Class models can be constrained, similarly to the way they are constrained using OCL. Class model constraints in FOML describe forbidden states (following PathLP). Constraints are checked offline, in a separate correctness testing, and not during regular runs.

An application that runs with a GPUAPI with some GPU card must run on an operating system that runs on a Computer with that GPU unit:

```
!- ?appl.applGApi[?gApi], ?gApi.apiGUnit[?Gpu],
   not (?appl.applOs[?Os], ?Os.osComp[?C], ?C.compG[?Gpu]);
```

The corresponding OCL constraint was shown in Section 2, just before the start of subsection 2.1.

A computer must have at least one operating system on which a "pdf" application runs:

```
!- ?c.os[?os], not (?os.osAppl[?appl],
   not ?appl.category["pdf"]);
```

An equivalent OCL constraint:

```
Context Computer
inv: self.os.osAppl.category->includes("pdf")
```

An object cannot be simultaneously an API and an operating system:

```
!- ?o:OS, ?o:ComputingAPI;
```

An equivalent OCL constraint:

```
Context OS
inv: OS.allInstances()->intersection
     (ComputingAPI.allInstances())-> isEmpty()
```

Intensional extension of object models:

Computer applications with a common category are alternative to each other:

```
?appl1.alt1[?appl2] :-
       ?appl1.category[?cat], ?appl2.category[?cat];
```

Alternative computer applications have a common category:

```
?appl1.category[?cat] :-
       ?appl1.alt1[?appl2],?appl2.category[?cat];
```

This rule implies the missing attribute link `Foxit.category["pdf"]` in the object model of Example 2, in Listing 2.

The alternative association is symmetric:

```
?appl1.alt1[?appl2] :- ?appl2.alt1[?appl1];
```

If an application runs on an operating system that runs on some computer, then the application is a software on that computer:

```
?C.softw[?Appl]:- ?C.os[?OS], ?OS.osAppl[?Appl];      (1)
```

If a computer application runs using a pair of GPU and Computing API element (the applGApi property), then it implements that API (the applApi property):

```
?appl.applApi[?compAPI] :-                            (2)
       ?appl.applGApi.gApi[?compAPI];
```

The latter rule implies the other missing link, `cuDNN.applApi[CUDA]` in the object model of Example 2. Without that, the instance is illegal, as the multiplicity constraint of property `applApi` is not satisfied for object `cuDNN`.

User rules that add derived (new) intensional properties or attributes:

If an application runs on an operating system that runs on a computer, then the application can be installed *on that computer:*

```
?appl.install[?C] :-                                  (3)
        ?appl.applOs[?Os], ?os.osComp[?C];
```

API objects that are linked to GPU objects can be classified as *graphics* APIs:

```
?api.graphics[true] :- ?api.apiGUnit[?];
?api.graphics[false] :- ?api: ComputingAPI, not ?api.apiGUnit[?];
```

4.2. Metalevel Modeling with FOML

The FOML meta-modeling capability enables specification of intensional structures of model elements, definitions of modeling meta-constraints, analysis of class and object models, and supporting general software engineering activities like testing and syntactic correctness validation. Below, we shortly describe the meta-modeling capabilities. The actual FOML system is described in Section 5.

4.2.1. Higher-order Intensional Elements of Models

FOML enables specification of new inductive, parameterized, intensional model elements, constructed on top of declared properties, associations and classes. The most useful ones are intensional parameterized properties, which are described using graph-inspired terminology of edges, paths, and cycles in model diagrams. FOML provides a library of higher-order constructors for such elements. We describe some such structures below, in order to provide a taste of this powerful capability.

Property composition: *objects ?o and ?v are related via the intensional property* `compose(?p1,?p2)` *if there is a "property path"* ?p1.?p2 *from ?o to ?v (or via* `compose(?p1,?mid,?p2)`*, if there is a "property path"* ?p1.?p2 *from ?o to ?mid to ?v):*

```
?o.compose(?p1,?mid,?p2)[?v] :-
   ?o.?p1[?mid].?p2[?v];
```

For example, rules (1) and (3) above can be rewritten more succinctly as follows:

```
?C.softw[?Appl]:- ?C.compose(os,osAppl)[?Appl];          (1')
?appl.install[?C] :- ?appl.compose(applOs,osComp)[?C];  (3')
```

Transitive closure: *The parameterized property* `closure(?p)` *describes the transitive closure of the reflexive property* ?p*:*

```
?o.closure(?p)[?v] :- ?o.?p[?v];
?o.closure(?p)[?v] :- ?o.?p.closure(?p)[?v];
```

The transitive closure can be used to identify *circularity* of reflexive properties, i.e., characterizing a circular path of *?p* related objects:

```
?p.circular[true] :- ?o.closure(?p)[?o];
```

For example, in the class model in Figure 1, association `partParent` between `Hardware` objects can be constrained to be non-circular:

```
!- part.circular[true];
```

A modeler can extend that class model with a new intensional association that computes the parts of a `Hardware` object:

```
?o.hardware_parts[?partslist] :-
   setof(?part,
       (?o:Hardware, ?o.closure(part)[?part]),
       ?partlist);
```

Here, `setof` is an aggregate operator, which collects all `?parts` that satisfy the condition `(?o:Hardware, ?o.closure(part)[?part])`, and returns the list of these `?parts`.

The metamodel of class models, which FOML uses for checking correctness of a user model (see Section 5), uses *closure* and *circular* to specify that class hierarchies are not circular. The metamodel includes the following constraint:

```
!- $subclass.circular[true];  % a forbidden state
```

Property composition and circularity can be extended to inductively defined property paths and cycles:

Paths parameterized property: ?*o and* ?*v are connected via a chain of properties*

```
?o.path([?p])[?v] :- ?o.?p[?v];
?o.path([?p|?chain])[?v] :- ?o.?p.path(?chain)[?v];
```

For example, a query for finding all objects accessible from *ThinkPad* in the object model in Figure 2:

```
?- ThinkPad.path(?path)[?o];
```

On the metalevel, a class model can be queried for property paths between classes. For example, Figure 1 can be queried for property paths that end in class `Computer`:

```
?- ?Class.path(?path)[Computer];
```

or for property cycles:

```
?- ?Class.path(?path)[?Class];
```

Moreover, class models can be restricted not to have property cycles:

```
!- ?Class.path(?path)[?Class];
```

4.2.2. Defining Meta-Constraints

FOML supports UML constraints using meta-classes, whose semantics is built into PathLP. We show here the encoding of the three constraints included in the class model of Figure 1: Generalization set, association class, and subsetting.

Generalization-set constraint: A generalization set constraint has the form

$\texttt{C.\$GS}(\texttt{C}_1,\ldots,\texttt{C}_n)[\texttt{kind}];$

where $n > 1$ and *kind* can be `disjoint, complete, overlapping, incomplete` or some subset of these. The semantics is subtyping $\texttt{C}_\texttt{i} :: \texttt{C}$, for $i = 1,\ldots,n$, plus the *kind* relation between any $\texttt{C}_\texttt{i}, \texttt{C}_\texttt{j}$, where $\texttt{i},\texttt{j} = 1,\ldots,\texttt{n}$.

```
?sub::?super :-
    ?super.$GS(?cls_lst)[?],?cls_lst._member[?sub];

?sub1.$disjoint[?sub2l] :-
    ?super.$GS(?cls_lst)[disjoint],
    ?cls_lst._member[?sub1],?cls_lst._member[?sub2],
    ?sub1 != ?sub2;
!- ?C.$disjoint[?D], ?o:?C,?o:?D;
```

Association class: An association class constraint has the form

`a.$assocClass[C];`

where `a` is an association and `C` its association class. The semantics requires a bijective mapping between objects of the association class and links of the association. The mapping is encoded by direct navigation from an association class object to the objects of its associated link:

In the class model:
```
gpuApi.$assocClass[GPUAPI];
```
In the object model:
```
NvidiaCUDA:GPUAPI;
NvidiaCUDA.gApi[CUDA];
NvidiaCUDA.apiGUnit[Nvidia];
```

The bijective restriction on association class mappings are expressed using these PathLP constraints:

A link of an association has a single corresponding object in its association class:

```
!- ?a.$assocClass[?AC], ?a.$assocProperty[?p1,?p2], ?o1.?p1[?o2],
   not (?ac:?AC, ?ac.?p1[?o1], ?ac.?p2[?o2]);
!- ?a.$assocClass[?AC], ?a.$assocProperty[?p1,?p2], ?o1.?p1[?o2],
   ?ac:?AC,  ?ac.?p1[?o1],  ?ac.?p2[?o2],
   ?ac':?AC, ?ac'.?p1[?o1], ?ac'.?p2[?o2],
   ?ac != ?ac';
```

An association class object has a single corresponding link in the association of that class:

```
!- ?a.$assocClass[?AC], ?a.$assocProperty[?p1,?p2], ?ac:?AC,
   not (?ac.?p1[?o1], ?ac.?p2[?o2]);
!- ?a.$assocClass[?AC], ?a.$assocProperty[?p1,?p2], ?ac:?AC,
   ?ac.?p1[?o1],  ?ac.?p2[?o2],
   ?ac.?p1[?o1'], ?ac.?p2[?o2'],
   (?o1 != ?o1' or ?o2 != ?o2');
```

In addition, an association class object implies the relevant associated link:

```
?o1.?p2[?o2] :-                                          (4)
   ?a.$assocClass[?AC], ?a.$assocProperty[?p1,?p2], ?ac:?AC
   ?ac.?p1[?o1], ?ac.?p2[?o2];
```

The last rule accounts for the "missing" link `Nvidia.gApi[CUDA]` in the FOML encoding of Figure 2, in Listing 2.

Subsetting: A subsetting statement has the form

```
p.$subsets[q];
```

where p, q are properties. The semantics is property subtyping, i.e., links of p are also links of q. The semantics can be encoded in FOML by the rule:

```
?o1.?q[?o2] :- ?o1.?p[?o2], ?p.$subsets[?q];
```

Using this rule, the object model encoding in Listing 2 infers the necessary link `ThinkPad.softw[linux]`.

4.2.3. Analysis of Models

FOML metamodeling base includes metaclasses like `$Class`, `$Attribute`, `$Association`, `$Property`, and accounts for their inter-relationships, dependencies, and constraints. Using these facilities, FOML can analyze and control class and object models. Here are some examples of model querying:

Find all properties whose minimum multiplicity is 1:

```
?- ?prop:$Property,
   ?SrcClass!?prop[?TrgClass]{1..?};
```

Find pairs of properties of the same association:

```
?- ?assoc.$assocProperty[?prop1,?prop2], ?prop1 != ?prop2;
```

Find reflexive properties:

```
?- ?Class!?prop[?Class];
```

Find classes that are accessible from class Computer:

```
?- Computer.path(?proplst)[?target];
```

Meta-facilities can be used to introduce a fine characterization of properties:

```
?p.kind[injective] :-
      ?p:$Property,inverse(?p).min[0],inverse(?p).max[1];
?p.kind[surjective] :-
      ?p:$Property,inverse(?p).min[1],inverse(?p).max[*];
?p.kind[bijective] :-
       ?p.kind[injective],?p.kind[surjective];
?o2.?p[?o1] :- ?p.symmetric[true],?o1.?p[?o2];
```

Other software engineering activities that analyze and test models, are described in the next section.

5. The FOML Tool

The FOML querying and verification tool (Khitron et al. 2017) is implemented in PathLP and also uses PathLP to specify class and object models as well as to query them. For example, both class model constraints like association class or property subsetting, and metamodel constraints, and model verification queries are expressed in PathLP. A modeling activity, like instance checking, which is largely based on the concept of well-typed instances from Section 3, is implemented as a set of PathLP constraints.

The underlying language of the PathLP subsystem of the FOML tool consists of the "pure" part, as described in Section 3, and for practical reasons it is augmented with support for arithmetic, aggregate functions (e.g., sum, count), I/O, and other useful builtins. It is also supported by various libraries for traversing graphs of linked objects and types, some written in PathLP and some in Prolog. The examples in Section 4.2.1 rely on these libraries.

The interface of the FOML tool provides several *contexts*, each tailored to a different software modeling activity that an end user might be engaged in.

1. **Meta-reasoning and analysis of class and object models:**
   - **Class models:** In this context, a user can load and then query the content of a class model. Analysis queries in this context are demonstrated in Sections 4.2.1 and 4.2.3. The meta-analysis of class models relies on an internal PathLP library that captures the inter-relationships and constraints, as described in Section 2.1. This library accounts also for implied elements and default values that are not explicitly declared, like missing association or property names. For example, if in Listing 1, in Section 4.1, the association name `partParent` is omitted, the FOML tool complements it as `$assoc(parent,part)`, or if property `gComp` is not specified, the tool adds it as `inverse(compG)`. Moreover, the tool adds missing default specifications, like `0..*` multiplicity constraints or `$Any` as the value type of an attribute. Meta-analysis of class models provides also *metric information* like size (e.g., number of classes), structure (e.g., property cycles in the model), and class-hierarchy structure (e.g., multiple inheritance).
   - **Object models:** This context supports similar analysis of an object model. That is, a user can load an object model for an already loaded class model, and query its content and structure. As we have already seen in querying the object model in Listing 2, in Section 4.1, the encoding does not declare implicit information. The meta-analysis context knows to infer implied data like implied links and object memberships.
2. **Querying class and object models:** In this context, a user can load an object model for an already loaded class model, and query inferred information. Querying and inferences in this context is demonstrated in Section 4.1.1.
3. **Verification and validation of class models:** In this context, a user can load an object model for an already loaded class model, and check whether the object model is a legal instance of the class model, i.e., satisfies all class model constraints. This context is for testing and validation. Negative tests, i.e., tests to find illegal instances, can be used to find constraint violations.
4. **Syntactic correctness:** This context enables checking the syntactic correctness of class and object models, i.e., checking whether a user model satisfies the constraints of its metamodel. To apply this context a user first loads the textual representation of a class model or of an object model for a syntactically correct class model.
   - **Class models:** A syntactically correct class model is a legal instance of the metamodel of class models, denoted MM_{CM}. To check this correctness the FOML

tool creates a representation of the class model as an instance of MM_{CM}. This object model representation is created as a PathLP model transformation from a class model to the concrete syntax of object models. Once this object model of MM_{CM} is created, it is validated in the above verification and validation context.

– **Object models:** In this context, simple meta rules like requirements that an object model includes only objects and links of classes and properties of the class model are checked. Similarly, object attributes should refer to appropriate class attributes. Yet, the syntactic check must rely on the meta-reasoning context, for inference of implied data that is not explicitly declared, as we have seen in Listing 2, in Section 4.1 .

The FOML tool can simultaneously support multiple class and object models. Each context can switch between models at will. This unique feature allows one to reason about several models simultaneously. For example, class models for different viewpoints of a domain can be queried for common classes, or agreement (or lack of) of common attribute types of common classes. This feature enables support in multilevel modeling. Indeed, there is in-progress project that aims to develop a multilevel modeling component for the FOML tool, along the lines of (Balaban et al. 2018).

Below, we present a sample session with the FOML tool. To help focus on the important, we remove some inessential chatter from the session. In this sample session, we assume that the class model of Listing 1 is in the file `computer.cls` and the object model of Listing 2 is in a file like `computer^model1.obj`. Commands entered by the user are shown in boldface, while mono-font is reserved for the chatter coming from the tool. The regular roman font is used for our in-line clarifications. For easier understanding, we use a menu-driven interface, which provides high-level functions of FOML, like loading and verification. To ask queries, we escape to the "expert" mode, which is essentially the PathLP command line mode. An experienced user can conduct the entire dialog via the expert mode.

```
cmd>  pathlp              ## start PathLP in OS command window

PathLP >  ?- foml;        %% start FOML
1 - load class model
2 - load class and object models
3 - check class model syntax
4 - check object model syntax
5 - check legality of instance
6 - list folder
7 - switch to FOML expert mode
8 - exit PathLP

foml >  1                 %% choose option 1
Class model file name: computer.cls
the current model is 'computer.cls'

foml>  7                  %% let's ask some class model queries

PathLP computer.cls >  ?- Computer!?prop[?type];
?prop = os                          %% answers
?type = OS

?prop = softw
?type = Software

?prop = part
?type = Hardware

?prop = tested(date)
?type = String

PathLP {computer.cls} >  ?- foml;   %% back to FOML menu interface

foml>  2                  %% now choose option 2
Class model file name: computer.cls
Object model file name: computer.obj
the current model is 'computer.obj'

foml>  5                  %% let's check legality
Checking object model 'computer.obj'
All constraints are satisfied.

foml>  7                  %% let's ask some object queries

PathLP {computer.obj} >  ?- ThinkPad.os[?X];
?X = Linux                          %% answers
?X = Windows

PathLP {computer.obj} >  ?- halt;  %% going home now

cmd>                      ## back to OS command window
```

6. Related Work

Development and study of software modeling frameworks and tools has been the focus of intensive research over the last decade. On the practical level, many frameworks, environments and tools have been developed and used, with the goals ranging from education, to academic research, to experimental and commercial tools. In industrial-strength software, we find professional modeling tools like EMF (Steinberg et al. 2008; EMF 2017) with its Papyrus modeling environment (IBM 2020a), RSA (IBM 2020b), Magicdraw (Magic 2020), and the Epsilon family of model management languages (Kolovos et al. 2008; Epsilon 2017). These systems support activities like model specification, investigation using model metrics, transformation and code generation.

Theoretical study of software models concentrate on formal aspects of their properties and management procedures. The need for semantics of class models has led to multiple approaches concerning desirable interpretations and extensions, including (1) translational approaches, mainly to logic (Berardi et al. 2005); (2) graph-based approaches (Kleppe & Rensink 2008), and direct set-based approaches (Calvanese et al. 1998; Balaban & Maraee 2013). The main theoretical questions concerning correctness of class models focus on issues of their *consistency* (Satoh et al. 2006), *finite satisfility* (Calvanese 1996; Balaban & Maraee 2013; Feinerer & Salzer 2013) and *simplification* (Feinerer et al. 2011; Taupe et al. 2016; Balaban & Maraee 2019).

While deciding correctness problems of UML class models

is hard (EXPTIME-complete (Berardi et al. 2005; Lutz et al. 2005; Artale et al. 2010)), it is nevertheless decidable. With the addition of OCL, all problems turn undecidable (yet EXPTIME-complete for the UML/OCL-lite fragment (Queralt et al. 2012)). UML/OCL applications usually rely on off-the-shelf solvers and theorem provers for validating and checking correctness of models. Most applications use bounded inference, i.e., size of model instances is restricted in advance.

UMLtoCSP (Cabot et al. 2007, 2014) translates UML/OCL models into Constraint Satisfaction Problems (CSP) for checking correctness properties, including consistency, bounded finite satisfiability, independence of invariants and instance completion. Alloy (Jackson 2002a, 2006), a relational-logic modeling language that is built on top of a SAT solver, is frequently used for verification of small models via instance generation and completion. Such applications rely on translation of UML/OCL specifications into Alloy (Anastasakis et al. 2010; Maoz et al. 2011). HOL-OCL (Brucker & Wolff 2008) is a theorem proving environment that supports interactive proofs of consistency, instance validation and query answering for UML/OCL models.

USE is a modeling environment that supports validation and verification of UML/OCL models (Gogolla et al. 2005; Gogolla et al. 2009; Kuhlmann et al. 2011). USE is also popular as an education system for teaching software modeling. Another popular educational system is Umple (Forward et al. 2012), which supports model-based code generation.

Logic programming-based approaches use logic rule-based encodings of class models for supporting inference and answer querying. In (Cali et al. 2012), $Datalog^{\pm}$ (Calì et al. 2012) is used for characterizing Lean UCD, a set of class models with a restricted subset of OCL constraints, for which conjunctive query answering of instances is tractable (measured in the size of instances). In (Malgouyres & Motet 2006), a syntax verification approach for UML models is described, which encodes UML metamodels in Constraint Logic Programming. Similar, but more general, verification is part of FOML (the syntactic correctness reasoning context, see Section 5). F-logic (Kifer, G., & Wu 1995), which lies at the origin of FOML, is used in (Igamberdiev et al. 2014, 2016; Neumayr et al. 2016) as a basis for multilevel modeling of software.

The FOML approach is quite different from those mentioned above. It is based on PathLP, which offers a simple, intuitive object-oriented syntax. It has a number of advantages and disadvantages with respect to SAT-based tools, like Alloy, which support OCL. The main disadvantage of FOML is that SAT-based tools can complete partial instances to full instances that satisfy all constraints. In contrast, FOML can only check if constraints are satisfied in a particular instance and therefore combining FOML with SAT-based tools is highly desirable. On the other hand, the strong points of FOML include support for meta-modeling, patterns, recursion, model querying, analysis, and testing. The model analysis and the testing features of FOML are made possible due to FOML being a Turing-complete executable logic language. Other logic programming based approaches to software modeling, like $Datalog^{\pm}$ (Calì et al. 2012), tend to sacrifice expressivity and functionality for gains in decidability and guarantees on efficiency.

7. Conclusion and Future Work

We have presented *FOML*, an expressive logic-based executable modeling language and tool (Khitron et al. 2017), that could benefit intelligent software modeling systems. FOML is designed as a conceptual layer on top of the *PathLP* path expression language. PathLP already has a functioning implementation, and most of FOML has been implemented and is usable for experimentation as well. Moreover, support for multilevel modeling is also on the way.

The FOML tool offers a unique combination of contexts for software modeling activities, including support for model level querying and inference, meta-level analysis, validation (testing), and syntactic correctness checking. It is unique in its ability to simultaneously handle multiple models, which opens up the possibility to support activities like model merging, comparison, and dealing with model inter-relationships.

A planned future development of the FOML tool involves extending the underlying PathLP language in the direction of HiLog and Transaction Logic (Chen et al. 1993; Bonner & Kifer 1998, 1994). Such extensions can add further flexibility to the PathLP language, and would enable support for behavioral models. In addition, we plan on adding a visual UI to the tool, preferably, via integration with an existing visual open source application for UML models.

We furthermore plan on extending the FOML tool by integrating it with other modeling software to complement the existing capabilities. One such possible tool is *FiniteSatUSE* (BGU Modeling Group 2018), which detects, identifies and provides advice for *finite satisfiability* problems in class models, and also performs optimization of multiplicity constraints (Balaban & Maraee 2019).

A different promising direction involves integration with software modeling tools like USE (Gogolla et al. 2005) or Umple (Forward et al. 2012), which also complement the functionality of FOML. USE is now integrated with SAT solvers that can determine satisfiability of a UML/OCL class model and support instance creation and completion (Kuhlmann et al. 2011). Umple adds the capability of code generation, and FOML can be integrated as a reasoning system, on top.

Acknowledgments

We would like to thank Azzam Maraee for helpful discussions. This work is supported in part by the BSF Grant 2017742 and NSF Grant 1814457.

References

Anastasakis, K., Bordbar, B., Georg, G., & Ray, I. (2010). On Challenges of Model Transformation from UML to Alloy. *Software and Systems Modeling*, *9*(1), 69–86.

Artale, A., Calvanese, D., & Ibánez-Garcıa, A. (2010). Full satisfiability of UML class diagrams. In *Proc. of the 29th int. conf. on conceptual modeling (er 2010).*

Atkinson, C., & Kühne, T. (2001). The essence of multilevel metamodeling. In *Uml* (pp. 19–33). Springer.

Atkinson, C., & Kühne, T. (2008). Reducing accidental complexity in domain models. *Software & Systems Modeling*, *7*(3), 345–359.

Balaban, M., Bennett, P., Doan, K. H., Georg, G., Gogolla, M., Khitron, I., & Kifer, M. (2016). A Comparison of Textual Modeling Languages: OCL, Alloy, FOML. In *16th international workshop on OCL and textual modeling, models.*

Balaban, M., Khitron, I., Kifer, M., & Maraee, A. (2018). Formal executable theory of multilevel modeling. In *Caise.*

Balaban, M., & Kifer, M. (2011). Logic-Based Model-Level Software Development with F-OML. In *Models 2011.*

Balaban, M., & Maraee, A. (2013). Finite Satisfiability of UML Class Diagrams with Constrained Class Hierarchy. *ACM TOSEM*, *22*(3), 24:1–24:42.

Balaban, M., & Maraee, A. (2017). *UML Class Diagram: Abstract syntax and Semantics.* https://goo.gl/UJzsjb.

Balaban, M., & Maraee, A. (2019). Removing redundant multiplicity constraints in UML class models. *Software & Systems Modeling*, *18*, 2717–2751. Retrieved from https://doi.org/10.1007/s10270-018-0696-z

Berardi, D., Calvanese, D., & Giacomo, D. (2005). Reasoning on UML class diagrams. *Artificial Intelligence*, *168*, 70-118.

BGU Modeling Group. (2018). *FiniteSatUSE – A Class Model Correctness Tool.* https://goo.gl/svXQwj.

Bonner, A., & Kifer, M. (1994, October). An Overview of Transaction Logic. *Theoretical Computer Science*, *133*, 205–265.

Bonner, A., & Kifer, M. (1998). A logic for programming database transactions. In J. Chomicki & G. Saake (Eds.), *Logics for databases and information systems* (pp. 117–166). Kluwer Academic Publishers.

Brucker, A., & Wolff, B. (2008). HOL-OCL: A Formal Proof Environment for UML/OCL. In *Fundamental approaches to software engineering* (Vol. 4961, p. 97-100). Springer-Verlag.

Cabot, J., Claris, R., & Riera, D. (2007). Umltocsp: a tool for the formal verification of UML OCL models using constraint programming. In *Ase 07, the twenty-second ieee-acm international conference on automated software engineering* (pp. 547–548). New York, NY, USA.

Cabot, J., Clariso, J., & Riera, D. (2014). On the Verification of UML/OCL Class Diagrams Using Constraint Programming . *Journal of Systems and Software*, *93*(0), 1 - 23.

Calì, A., Gottlob, G., & Lukasiewicz, T. (2012). A general datalog-based framework for tractable query answering over ontologies. *Web Semantics: Science, Services and Agents on the World Wide Web*, *14*, 57–83.

Cali, A., Gottlob, G., Orsi, G., & Pieris, A. (2012). Querying UML Class Diagrams. In *Foundations of software science and computational structures* (p. 1-25). Springer Berlin Heidelberg.

Calvanese, D. (1996). Finite model reasoning in description logics. In *The 5th int. conf. on the principles of knowledge representation and reasoning (kr-96).*

Calvanese, D., De Giacomo, G., Lenzerini, M., Nardi, D., & Rosati, R. (1998). Description logic framework for information integration. In *6th intl. conf. on the principles of knowledge representation and reasoning (kr'98)* (p. 2-13).

Chen, W., Kifer, M., & Warren, D. (1993, February). HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, *15*(3), 187–230.

Deutsch, A., Sui, L., & Vianu, V. (1999). Xml path language (xpath) version 1.0. w3c recommendation, the world wide web consortium. In *Journal of computer and system sciences (jcss) 2007; 73(3):442–474.*

EMF. (2017). *Eclipse modeling framework (emf).* https://www.eclipse.org/modeling/emf.

Epsilon. (2017). *Epsilon.* https://www.eclipse.org/epsilon/.

Feinerer, I., & Salzer, G. (2013). Numeric Semantics of Class Diagrams with Multiplicity and Uniqueness Constraints. *Software and Systems Modeling (SoSyM).*

Feinerer, I., Salzer, G., & Sisel, T. (2011). Reducing Multiplicities in Class Diagrams. In *Model driven engineering languages and systems* (pp. 379–393).

Forward, A., Badreddin, O., Lethbridge, T. C., & Solano, J. (2012). Model-driven rapid prototyping with umple. *Software: Practice and Experience*, *42*(7), 781-797.

France, R., & Rumpe, B. (2007). Model-Driven Development of Complex Software: A Research Roadmap. In *International conference on software engineering* (pp. 37–54).

France, R. B., Ghosh, S., Dinh-Trong, T., & Solberg, A. (2006). Model-Driven Development Using UML 2.0: Promises and Pitfalls. *Computer*, *39*, 59-66. doi: http://doi.ieeecomputersociety.org/10.1109/MC.2006.65

Frankel, D. (2003). *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley-India.

Frohn, J., Himmeröder, R., Lausen, G., May, W., & Schlepphorst, C. (1998). Managing semistructured data with FLORID: A deductive object-oriented perspective. *Information Systems*, *23*(8), 589–613.

Gogolla, M., Kuhlmann, M., & Hamann, L. (2009). Consistency, Independence and Consequences in UML and OCL Models. In *Proceedings of the 3rd international conference on tests and proofs* (pp. 90–104). Springer-Verlag.

Gogolla , M., Bohling, J., & Richters, M. (2005). Validating UML and OCL models in USE by automatic snapshot generation. *Journal on Software and System Modeling*, *4*, 386-398.

Henderson-Sellers, B. (2012). *On the mathematics of modelling, metamodelling, ontologies and modelling languages*. Springer Science & Business Media.

IBM. (2020a). *Eclipse papyrus.* https://www.eclipse.org/papyrus/.

IBM. (2020b). *Ibm rational software architect designer.* https://www.ibm.com/developerworks/downloads/r/architect/index.html.

Igamberdiev, M., Grossmann, G., Selway, M., & Stumptner, M. (2016). An integrated multi-level modeling approach for industrial-scale data interoperability. *Software & Systems Modeling*, 1–26. Retrieved from http://dx.doi.org/10.1007/s10270-016-0520-6 doi: 10.1007/s10270-016-0520-6

Igamberdiev, M., Grossmann, G., & Stumptner, M. (2014). An Implementation of Multi-Level Modelling in F-Logic. In *1st international workshop on multi-level modeling (multi 2014)*

(pp. 33–42).
Jackson, D. (2002a). Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *11*(2), 256–290.
Jackson, D. (2002b). Alloy: A new technology for software modelling. In *Tacas '02*.
Jackson, D. (2006). *Software Abstractions: Logic, Language and Analysis*. The MIT Press.
Kern, H., & Kuhne, S. (2007). Model interchange between aris and eclipse emf. In *7th oopsla workshop on domain-specific modeling at oopsla* (Vol. 2007).
Khitron, I., Balaban, M., & Kifer, M. (2017). *The FOML Site*. https://goo.gl/AgxmMc.
Khitron, I., Kifer, M., & Balaban, M. (2011a, November). An Overview of PathLP: A Logic Programming Language of Path Expressions. In *Ibm programming languages and development environments seminar.* Haifa, Israel.
Khitron, I., Kifer, M., & Balaban, M. (2011b). *PathLP: A Path-oriented Logic Programming Language.* The PathLP Web Site. (http://pathlp.sourceforge.net)
Kifer, M. (2007). *FLORA-2: An object-oriented knowledge base language.* The FLORA-2 Web Site. (http://flora.sourceforge.net)
Kifer, M., G., L., & Wu, J. (1995). Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, *42*(4), 741–843.
Kifer, M., Kim, W., & Sagiv, Y. (1992, June). Querying object-oriented databases. In *Acm sigmod conf. on management of data* (p. 393-402). NY: ACM.
Kifer, M., Lausen, G., & Wu, J. (1995, July). Logical foundations of object-oriented and frame-based languages. *Journal of ACM*, *42*, 741–843.
Kleppe, A., Warmer, J., & Bast, W. (2003). *Mda explained: The model driven architecture(tm): Practice and promise*. Addison-Wesley Professional.
Kleppe , A., & Rensink, A. (2008). On a graph-based semantics for UML class and object diagrams. In C. Ermel, J. D. Lara, & R. Heckel (Eds.), *Graph transformation and visual modeling techniques* (Vol. 10). EASST.
Kolovos, D., Paige, R., & Polack, F. (2008). The epsilon transformation language. In *In international conference on model transformation.*
Kuhlmann, M., Hamann, L., & Gogolla, M. (2011). Extensive Validation of OCL Models by Integrating SAT Solving into USE. In *Tools europe 2011* (Vol. 6705, p. 290-306). Springer.
Lloyd, J. (1987). *Foundations of logic programming (second edition)*. Springer-Verlag.
Lutz, C., Sattler, U., & Tendera, L. (2005). The complexity of finite model reasoning in description logics. *Information and Computation*, *199*, 132-171.
Magic, N. (2020). *Magicdraw.* https://www.nomagic.com/.
Malgouyres, H., & Motet, G. (2006). A uml model consistency verification approach based on meta-modeling formalization. In *Proc. acm symp. on applied computing* (p. 1804-1809).
Maoz, S., Ringert, J., & Rumpe, B. (2011). CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In J. Whittle, T. Clark, & T. Kühne (Eds.), *Model driven engineering languages and systems* (Vol. 6981, pp. 592–607). Springer-Verlag.
Maraee, A., & Balaban, M. (2012). Inter-association Constraints in UML2: Comparative Analysis, Usage Recommendations, and Modeling Guidelines. In *Models 2012*.
Neumayr, B., Schuetz, C. G., Jeusfeld, M. A., & Schrefl, M. (2016). Dual deep modeling: MLM with dual potencies and its formalization in F-Logic. *SoSyM*, 1–36. Retrieved from http://dx.doi.org/10.1007/s10270-016-0519-z doi: 10.1007/s10270-016-0519-z
Object Management Group (OMG). (2012). *Object Constraint Language (OCL)* (Specification No. Version 2.3.1). OMG. Retrieved from http://www.omg.org/spec/OCL/2.3.1/PDF/
OMG. (2017). *UML 2.5.1* (Specification No. Version 2.5.1). http://www.omg.org/spec/UML/2.5.1/PDF.
Queralt, A., Artale, A., Calvanese, D., & Teniente, E. (2012). OCL-Lite: Finite Reasoning on UML/OCL Conceptual Schemas. *Data & Knowledge Engineering*, *73*, 1 - 22.
Satoh, K., Kaneiwa, K., & Uno, T. (2006). Contradiction finding and minimal recovery for UML class diagrams. In *The 21st ieee intl. conf. on automated software engineering* (pp. 277–280).
Schmidt, D. (2006). Model-driven engineering. *IEEE computer*, *39*(2), 25–31.
Sendall, S., & Kozaczynski, W. (2003, sep.). Model transformation: the heart and soul of model-driven software development. *Software, IEEE*, *20*(5), 42 - 45. doi: 10.1109/MS.2003.1231150
Steinberg, D., Budinsky, F., Merks, E., & Paternostro, M. (2008). *Emf: eclipse modeling framework*. Pearson Education.
Swift, T., & Warren, D. (2011). Xsb: Extending the power of prolog using tabling. *Theory and Practice of Logic Programming*.
Taupe, R., Falkner, A., & Schenner, G. (2016). Deriving tighter component cardinality bounds for product configuration. In *18th international configuration workshop, (8 pages).*
Van Gelder, A., Ross, K., & Schlipf, J. (1991). The well-founded semantics for general logic programs. *Journal of ACM*, *38*(3), 620–650.
Warmer, J., & Kleppe, A. (2003). *The object constraint language: Getting your models ready for mda*. Addison-Wesley Publishing Co., Inc.
Yang, G., & Kifer, M. (2003). Inheritance in rule-based frame systems: Semantics and inference. *Journal on Data Semantics*, *2800*, 69–97.

About the authors

Mira Balaban is a Professor Emerita in the Computer Science Department at Ben Gurion University, Israel. She is also a graduate of music performance from the Rubin Academy of Music in Tel-Aviv. Her research interests include software modeling, with emphasis on correctness, optimization, languages and inference of models, programming languages, and Computer Music. Contact her at mira@cs.bg.ac.il.

Igal Khitron is a Ph.D. candidate in the Department of Computer Science at the Ben Gurion University in Israel. Contact him at khitron@cs.bg.ac.il.

Michael Kifer is a Professor with the Department of Computer Science, Stony Brook University, USA. His work spans the areas of knowledge representation and reasoning (KRR), logic programming, Web information systems, and databases. He published four text books and numerous articles in these areas as well as co-invented F-logic, HiLog, Annotated Logic, and Transaction Logic. In 1999 and 2002, Kifer was a recipient of the prestigious ACM-SIGMOD "Test of Time" awards for his works on F-logic and object-oriented database languages and in 2013 he received a 20-year "Test of Time" award from the Association for Logic Programming (ALP) for his work on Transaction Logic. Kifer is also a recipient of Chancellor's Award for Excellence in Scholarship. To contact the author, visit http://www.cs.stonybrook.edu/~kifer.