Ontology-Oriented Programming: Static Typing for the Inconsistent Programmer*

Neil M. Goldman

Teknowledge Corporation
4640 Admiralty Way, Suite 1001
Marina del Rey, CA 90292 USA
ngoldman@teknowledge.com

Abstract. Ontologies and object-oriented data models differ little in their *declarative characterization* of a domain. Differences in the *application* of these models, however, has led to quite different characteristics of the programming languages used to create and manipulate them – notably in the presence or lack of static typing.

Some ontology-reliant applications need to work with models that are *inconsistent* with the underlying ontology's declarative norms. Object-oriented languages preclude construction of such models. A secondary distinction is that ontologies, at least as used in the description logic community, provide a semantics that requires an *implicit* interpretation of extensional data, where possible, that retains consistency with the declarative norms, whereas object-oriented languages treat the norms as constraints that must be *explicitly* satisfied by a model.

An *ontology compiler*, currently targeting the Microsoft .Net language family, produces a traditional object-oriented class library that captures the declarative norms of an ontology. It enhances this library with additional methods that allow construction of models that are inconsistent with these norms. An application, rather than the library itself, determines when an operation should raise exceptions. *Annotations* to an ontology allow for tradeoffs between the flexibility of the generated library and its performance.

1 Introduction

Semantic annotations of web data will be both created and consumed by software acting on behalf of people or on behalf of other software agents. This paper is concerned with the nature of the programming languages and environments in which programmers compose such software. Some of these programs will be *generic* in nature – independent of any particular ontology. Validity checkers and query servers [1] fall into this category. Many more applications – e.g., a personal agent to manage bidding for an item at an online auction site – will require software that is written to deal with classes and properties of a specific ontology.

* This research was supported by DARPA contract F30602-00-C-0202.

D. Fensel et al. (Eds.): ISWC 2003, LNCS 2870, pp. 850–865, 2003.

The current generation of ontology based programming tools provides only weak support for programming the ontology-specific class of applications. Better support could be provided by *automatic generation* of software class libraries based on the content of the ontologies needed for such an application.

Section 2 describes the considerable overlap, as well as the few significant differences, between *object-oriented* (Ob-O) and *ontology-oriented* (On-O) data modeling. Section 3 looks at a specific, and I believe representative, example of applying a tool designed for implementing generic applications to the job of building an ontology-specific application, focusing on the absence of benefits afforded by static typing. Section 4 outlines a particular mapping from ontology-oriented models, as exemplified by the *OWL* family, to object-oriented class libraries, as exemplified by Microsoft's *.Net Framework*.

There may be more than one useful mapping from an ontology to a class library for manipulating the models allowed by that ontology. Section 5 illustrates several *annotations* that afford control over tradeoffs between flexibility and performance of the generated class library.

The objective of this technique is to support implementation of ontology-specific applications as depicted in Fig. 1, by automating the generation of ontology-specific class libraries for traditional object-oriented programming languages.

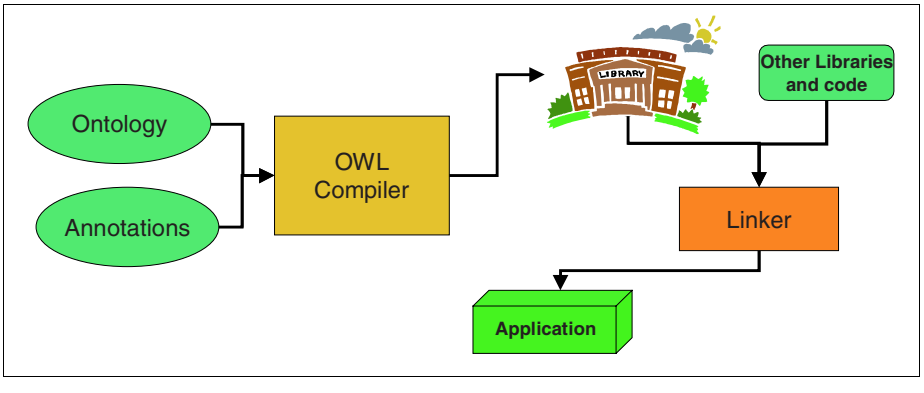

Fig. 1. Ontology-specific Application Development. An *ontology* and annotations are *compiled* to produce an ontology-specific *library*. This library is *linked* with other libraries and code to produce an *application*

2 On-O vs. Ob-O Data Modeling

Ontology languages, such as *OWL*[2], and object-oriented languages such as Microsoft's *.Net Framework* characterize a domain of discourse in quite similar fashion. In both we find a fundamental partitioning of the domain into *individuals* (**objects**) and *literals* (**values**). The universe of *individuals* (**objects**) is categorized into *classes* (**reference types**). The universe of *literals* (**values**) is categorized into *datatypes* (**value types**). OWL provides *ObjectProperties* to relate pairs of individuals, and *DatatypeProperties* to relate individuals to literals. .Net provides

fields[1] of **objects** for both purposes, but each field is used uniformly to relate an object to *either* another object *or* to a value, and so we could accurately partition them into **ObjectFields** and **DatatypeFields**.

Both *ObjectProperties* and *DatatypeProperties* can be given *domains* and *ranges*, restricting the individuals and literals that may be related by the properties. Every **field** also has a domain – some reference type – and a range.

Finally, both the On-O and Ob-O modeling languages provide intensional requirements of subsumption and disjointness[2] – that is, statements or implications of subsumption and disjointness that must hold for *all* model instances. The Ob-O reference types include both **class types** and **interface types**. It is the interface types that bear a considerable similarity to the *classes* of On-O models – in particular, if I_1 and I_2 are interface types, then I_1 may "inherit"[3] from I_2 (and thus be subsumed by it). Alternatively, if I_1 and I_2 have no common subtype, then they must be disjoint.

Another area of common concern is characterizing the number of *individuals* (**objects**) that may be related to a given *individual* (**object**) by a particular *ObjectProperty* (**field**). In particular, both On-O and Ob-O paradigms encourage a modeler to distinguish functional relationships (at most one range individual for a given domain individual) from others. The On-O paradigm cleanly isolates this issue, providing **restrictions** that allow cardinality restrictions other than "at most one" to be stated. The Ob-O paradigm confounds the characterization of cardinality with that of range. When a **field** is declared to have a reference type (e.g., Person) as its range, this means "at most one Person".[4] A programmer accommodates multiple range values by specifying a collection type (e.g., "array of Person") as the **field**'s range.[5]

The final commonality worth mentioning is that all the declarative information about classes (**reference types**), *datatypes* (**value types**), and *properties* (**fields**) in both paradigms is available in models of their own. This facility (referred to as **reflection** on Ob-O languages) is the basis for writing generic applications.

Although the commonalities are considerable, the Ob-O and On-O modeling paradigms are not isomorphs of one another. Each provides means of saying some things that the other cannot. These differences reflect the ways in which these two paradigms have been traditionally used. In particular, On-O models [3] evolved with the expectation that they would be consumed by software that supplies sophisticated (relative to traditional software, at least) *inference* capabilities. Ideally, such software makes no distinction, other than performance, between a model where a statement is

---

[1] Software engineering concerns – primarily encapsulation – lead Ob-O languages to provide **methods** (and, in .Net, **properties**) in addition to Fields. However, **fields** capture the fundamental modeling of relationships in the domain.

[2] *Owl Lite* does not provide a way to declare classes to be disjoint, but other *Owl* variants and description logic languages do provide this.

[3] "Inherit" in this context is the imposition of a requirement that any class that implements I_1 must also implement I_2.

[4] Ob-O languages traditionally support a Null object that can be stored in any reference-typed **field**. With very rare exceptions, use of the Null value is the means of saying "no range values for this domain object". There is no analog to the Null value for value types, so use of a value type as a **field** range means "exactly one."

[5] These collection types commit to more than just "multiple objects". They often commit to an ordering of the objects, or to allowing a bag rather than just a set.

explicit and one where the statement not explicit but is a logical consequence of other statements and the ontology. For example, if the *ObjectProperty* "hasNationality" is stated to have the *Class* "Country" as its range, and the statement "x hasNationality y" appears in a model, then the individual y has "Country" as one of its classes in the model, regardless of whether the fact "y hasClass Country" was ever explicitly added to the model. Designers of On-O modeling languages like *Owl* carefully limit the logical expressiveness of the language so that such expectations are plausible.

Ob-O modelers, on the other hand, do not expect much inference to be performed by the automatically generated software that implements their models. They do expect the software to enforce the logical implications of their data modeling, but to do by restricting what models can be built. In particular, models are built incrementally and an increment is not allowed if the result would be inconsistent. In the Ob-O world, the increment "x hasNationality y" would be rejected unless "y hasClass Country" were already present. One significant exception to this is supertype inference. The Ob-O modeler *does* expect "y hasClass PoliticalUnit" to be inferred whenever "y hasClass Country" is explicit (or inferable), assuming Country was declared to be a subtype of PoliticalUnit. Ob-O programming languages like Java[4] and the .Net[5] family actually exclude inconsistent models primarily through static typing, rejecting any program that could build an inconsistent model, rather than through runtime consistency checks. These languages provide *safe casts* to deal with circumstances where static type reasoning is too weak to ensure compliance. Programs that use reflection to build models are also subject to runtime exceptions from increments that would lead to inconsistent models.

The point of this discussion is that the On-O and Ob-O paradigms differ little in how they characterize a domain of discourse, but differ significantly in the anticipated use of that characterization by software. The one fundamental difference originates in the means by which *individuals* (**objects**) are categorized into *classes* (**reference types**). In the On-O paradigm, an individual may be explicitly (or implicitly) categorized into multiple classes. This *might* create an inconsistent model – e.g., if a pair of these classes had been declared disjoint in the ontology – but in general it is not a problem. In the Ob-O paradigm, however, each object belongs to a *single* most-specific, named, **class type**, and of course to any **class** or **interface** types that subsume it. As a consequence, the Ob-O data modeler must *anticipate* all possible intersections of types that may be needed by applications.

A second difference we shall have to confront in generating a class library suitable for manipulating models based on an ontology is the notion of identity. Owl has adopted from RDF the practice of using a URI as a globally shared "name" for an individual. While it is presumed that all uses of such a name refer to the same individual, it is *not* presumed that distinct names refer to distinct individuals. A given ontology and model will often justify neither the conclusion that the names refer to the same individual nor or that they refer to distinct individuals. In languages used to manipulate Ob-O models an individual is added to a model by applying a **New** operator, and each such individual is distinct, insofar as the language's innate reference equality semantics is concerned, from every other individual. As a result, an Ob-O model cannot contain both "a person named Will Shakespeare" and "a person named Francis Bacon" without making a commitment as to whether they are the same

or distinct individuals. On-O models, on the other hand, have the option of stating that they are the same, stating that they are different, or reminaing noncommittal. [6]

3 Generic vs. Ontology-Specific Programming

If we accept that the primary value of the Semantic Web will come from software applications that can create and reason about ontology-based models, we need to ask whether existing programming languages are well-suited to building these applications.

The kinds of applications that are generally suggested for the Semantic Web seem to fall into two very different camps. Some applications – perhaps better thought of as tools – are generic in that their construction requires knowledge of the meaning of terms used in creating ontologies, such as 'subClassOf', but requires no knowledge of terms or semantics from any specific ontology. The kinds of tasks performed by such tools are things like consistency checking and query answering.

A second kind of application is one that deals with models based on one or more specific ontologies. [6] posits an agent that finds a suitable physician for a patient. This agent is concerned with some medical terms and some distance and time concepts. There are two reasons why non-generic applications are necessary:

- Some agents will need to supply semantics that cannot be expressed in the ontology language. Recall that ontology languages purposely limit their expressivity to retain decidability. The day may come when the additional semantics can be expressed in a standardized "declarative" language; it is possible today to supply them with procedural code.
- Even where the semantics expressed in an ontology is *logically* sufficient for an application, performance considerations will often dictate a non-generic implementation.

One approach to building Semantic Web tools and applications that has been popular is to rely on class libraries for mainstream object oriented languages (Java, .Net). The class library provides classes and fields that hold both ontologies and models built from them. Import/export methods provide a means to map between this representation and an XML-based "interchange" representation. The library provides two additional capabilities:

- a limited amount of inference, following the semantics expressed by the loaded ontology. Which of the authorized inferences are provided seems to be an ad-hoc implementation decision.
- a reflection-like set of methods that allow a programmer to index into a model using the types and properties of the ontologies on which it is based.

Fig. 2 exhibits the use of such a library to write a fragment of the sort of code that would be used in an ontology-specific application. The code uses the *Jena*[7,8] class library for Java, and is taken directly from a tutorial provided by the authors of that library. Jena is a library that can manipulate pure RDF models as well as models that use the DAML+OIL[9] extensions. The encoded algorithm is examining a model

[6] Typically Ob-O languages predefine a coarser-grained *Equals* equivalence operation that can be overridden on a class-by-class basis. But *Equals*, like reference equality, is boolean-valued; it does not provide "maybe" as a possible result.

based on the VCARD ontology[7], attempting to determine the name of a person whose email address is "amanda_cartwright@example.org".

```
DAMLModel model = … // code that loads the VCARD ontology
                    // and some data based on that ontology
DAMLClass vcardClass =
    (DAMLClass)model.getDAMLValue(vcardBaseURI+"#VCARD");
DAMLProperty fnProp =
   (DAMLProperty) model.getDAMLValue(vcardBaseURI+"#FN");
DAMLProperty emailProp = (DAMLProperty)
                  model.getDAMLValue(vcardBaseURI+"#EMAIL");
Iterator i = vcardClass.getInstances();
while (i.hasNext()) {
  DAMLInstance vcard = (DAMLInstance) i.next();
  Iterator i2 =
      vcard.accessProperty(emailProp).getAll(true);
  while (i2.hasNext()) {
     DAMLInstance email = (DAMLInstance) i2.next();
     if (email.getProperty(RDF.value).getString().equals(
                   "amanda_cartwright@example.org" ) ) {
       DAMLDataInstance fullname =
        (DAMLDataInstance) vcard.accessProperty(fnProp).
                           getDAMLValue();
       if ( fullname != null )
        System.out.println("Name: "+ fullname.getValue().
                                    getString());
     }
  }
}
```

Fig. 2. Jena code used for an ontology specific application

Several points stand out in an examination of this code.

- The Java types of the program's variables (**DAMLModel**, **DAMLClass**, **DAMLProperty**, **DAMLInstance**, **DAMLDataInstance**) are analogous to reflection types. They are *not* the classes introduced by the VCARD ontology. They *are* classes a programmer would want for writing generic applications.
- The names of classes and properties from the VCARD ontology appear as string literals (*VCARD*, *FN*, *EMAIL*).
- Datatype values stored in the model must be explicitly converted (getString) to an appropriate Java value type.

Fig. 3 displays the same algorithm, as it would appear in an object-oriented language using a VCARD class library. This example uses the syntax of VisualBasic.Net, but the code would isomorphic or nearly so in C# or even Java. In this case:

- The variable types (**VCARD, String**) and method names (*vcards*, *email*, *fn*) are those introduced or used by the VCARD ontology itself, not reflection types.
- Values of datatypes (in this example, strings) appear to the programmer as values from the value types of the embedding language.[8]

[7] http://www.hpl.hp.com/semweb/iswc2002/JenaTutorial.Alpha/DAML/solutions/vcard-daml.rdf

```
Dim model as VcardOntology.model
Dim vc as Vcard
for each vc in model.vcards
  dim eaddr as String
  for each eaddr in vc.emails
    if eaddr.equals("amanda_cartwright@example.org") then
      dim fullname as String = vc.fn
      if not fullname is Nothing then
         Console.Writeline("Name: " & fullname)
      end if
    end if
  next eaddr
next vc
```

Fig. 3. Statically typed code for an ontology specific application

The code in Fig. 3 is objectively more concise, and subjectively easier to comprehend, than that of Fig. 2. Programming in a statically typed environment conveys other important benefits. First, numerous programming errors that would only be detected by debugging in the Jena code will be caught by the compiler in the .Net code. These range from simple typographical errors, such as using "*#VCRD*" or "#vcard" instead of "#VCARD", to semantic errors that show up as type mismatches, such as using "#Orgname", a property whose domain is **Orgproperties**, rather than "VCard", in place of "#FN", a property whose domain is **Vcard**. In fact, such errors are often detected prior to compilation, or avoided entirely, in modern IDEs for statically typed languages, since programmers are offered menus and automatic name completion restricted to type-correct choices. Finally, the compilers of statically typed languages take advantage of the statically declared data model to produce smaller and more efficient code.

So far we have only observed that a *programmer* could take the same domain expertise required to write an ontology and could himself write an Ob-O class library that was more suitable for implementing many, if not all, ontology-specific applications for that domain. This is not surprising. The discussion in Section 0 goes beyond this, arguing that a well-written ontology already contains a declarative representation of the knowledge needed to construct that library – i.e., the library can be *generated* from the ontology automatically. The generated library can easily retain the correspondence between its types and fields and the corresponding concepts in the ontology as a mapping to the concept's URIs. Making this mapping available to an Ob-O language's reflection facilities enables a single *generic* program to import existing models encoded in their interchange syntax (RDF/XML) into the class library representation, and to export models created within the class library in interchange syntax. Thus no additional code needs to be written, or even generated, for the class libraries to share data with other web agents.

[8] Technically "string", unlike numeric types, is a reference type in the .Net languages as well as in Java. However, since strings have no mutable state, and provide a value-based equality operator, "string" acts for all practical purposes as an optimized value type.

Not so Fast…

Just what is the relationship between the Jena code in Fig. 1 and the .Net code in Fig. 3? Will they produce the same results across all possible model instances? Jena, after all, is capable of storing even inconsistent models. Furthermore, these code snippets contain no code that actually builds or increments models. Can the .Net statically typed library construct all possible models that could be constructed by the Jena code?

The answers to these questions depend on the issues of consistency and inference alluded to in section 851. A model, consisting of a finite set of assertions P(d,r) is *inconsistent* only if the negation of one of those assertions can be *inferred* from the model and the ontology. Since negations cannot be explicitly asserted, this must arise from an inferable negation. In the case of the Owl family this will involve at least one of the following:

- asserting that two individuals are distinct
- declaring that a property is functional (enabling inference of distinctness)
- declaring that two classes are disjoint
- declaring that one class is the complement of another (entailing their disjointness)

Many useful ontology specific applications do not rely on the ability to represent inconsistent models. Other suggested applications – particularly those that attempt to bring together data about the same domain of discourse from disparate sources – do seem likely to need to hold inconsistent models.[9]

At first blush the notion of static typing may seem incompatible with the ability to hold inconsistent models. In fact they are not incompatible. This will be addressed in greater detail in the following section of the paper. For now, simply observe that an Ob-O language can readily provide multiple methods that read and write the same data storage. By using run-time casts in their implementation, these methods can provide *different* views regarding the type of the storage. For example, what appears to be an "array of Thing" when viewed through some methods may appear as an "array of Person" when view through alternative methods. What allows this to work is that some of the casts used in the "array of Person" view will raise exceptions in the case that a non-Person has been added to the array through the "array of Thing" view. In summary, there is no reason why a class library cannot simultaneously provide a statically typed view of a model and simultaneously provide a statically typeless view of the same model. The typeless view would be capable, just like the Jena library, of holding an arbitrary set of explicit assertions.

Inference, on the other hand, is the ability to materialize assertions that are entailed by as well as those that are explicit in a model. Since any entailed assertion *could have been* explicitly asserted, inference does not introduce any added problems for the statically typed view of the model.

[9] In this regard, it is of concern that the interfaces through which people author ontologies and models to date seem to closely parallel the underlying language primitives, making it likely that desirable assertions of distinctness and declarations of disjointness will be inadvertently omitted.

4 Generating an Ob-O Class Library from an Ontology

The class library generator accepts as input an ontology (currently, DAML+OIL) and produces as output a .Net class library. It produces this library both in its binary form (an assembly) and in source code form (both VisualBasic and C#). Production of multiple source languages is handled by .Net's object model for code, which allows code generators to create abstract program models which can be both compiled and translated to the various concrete .Net languages.

The scheme by which this generator works was outlined above. The strategy is to provide pairs of typed and untyped fields for storage. Each typed field is associated with a .Net type T and holds a list of references each of which is a .Net instance of T – that is, **`TypeOf r is T`** would be true for every reference r in the list. T is a type in the generated library that serves as the "implementation" of a class in the ontology. The paired untyped field will hold references that need to be treated as instances of that type, but whose .Net type is not consistent with it. We will soon see where such instances arise. Corresponding to each such field pair, the library will provide one set of properties and methods that provide a statically typed view of the model and raise an exception if the untyped field is non-empty. A second set of properties and methods provides a typeless view that treats the concatenation of the pair of fields as a single collection of "Thing"s. The typeless view supports (portions of) applications that cannot abide the requirements of static typing.

In generating the types for a class library's fields, properties, and methods only *global* domain and range restrictions are taken into account. Implications about domain class that are conditional on range class and vice versa cannot be enforced statically and thus play no role in the assignment.

The descriptions below will refer to the accessibility of fields, methods, and properties as simply Public or Private. However, in many cases what is listed as Private actually uses the accessibility scope (called *Friend* in VisualBasic or i*nternal* in C#) that allows an element to be accessed from elsewhere in the library containing the element but not from outside that library.

Mapping Individual Classes

Suppose the ontology is named O. The class library will contain a public class **O.Model**. For each non-anonymous class C in O, the class library will contain a private class **O.C**. **O.C** will have a zero-argument constructor. The class **O.Model** will have a private field **my_Cs**, of type ArrayList[10]. This will hold only objects created by the constructor of **O.C**. The initial value is the empty list. A public method **O.Model.NewC()** implements allocation of new instances of **O.C**, calling its constructor and adding the result to **my_Cs**.

[10] This is an artifact of the absence of templates in the .Net languages. The only directly supported statically typed collection of Xs is "X()" (array of X), but these are not growable like ArrayLists. Thus we use a more flexible collection type and pay the price of a run-time cast (in the library code) each time an item is obtained from the list.

The library will also have a public interface **O.IC. O.C** will implement **O.IC.** We need the interface, as well as the class, because *multiple* inheritance only exists for interface types, not class types.

Mapping Subclassof

Suppose that the closest non-anonymous superclasses of C in O are $\{Sup_1 \ldots Sup_j\}$. Then for each $1 \leq i \leq j$, we will have **O.IC** inherits $\mathbf{O.ISup_i}$. The net effect is that **O.C** now implements its direct interface, **O.IC**, which in turn inherits each interfaces paired with a superclass. Since this pattern is followed for *every* class, **O.C** is forced to implement the interface of every class in its superclass hierarchy. Fig. 4 depicts this mapping in a simple case.

Suppose also that the closest non-anonymous subclasses of C in O are $\{Sub_1 \ldots Sub_n\}$. There will be a public method **O.Model.EnumerateCs** that retuns a statically typed enumerator of **O.C**s. This enumeration will include objects in **my_Cs**, as well as objects produced by $\mathbf{EnumerateSub_1s}$, etc. In other words, it will include objects created by the constructor of **O.C** or by the constructor of any of the classes corresponding to subclasses of C. In the case of DAGs, of course, this enumerator does *not* enumerate a class multiple times.

Mapping ObjectProperty and DatatypeProperty

Let $\mathcal{P}$(C) be the set of ObjectProperties in O whose global domain is declared to be C or a (not necessarily direct) superclass of C. If p is an ObjectProperty with no global domain declaration, then the domain is Thing, and we include p in $\mathcal{P}$(C).[11] In other words, $\mathcal{P}$(C) contains those properties applicable to *all* instances of C. For each $p \in \mathcal{P}$(C), we give **O.C** a private field **O.C.myps**. The field is assigned the type ArrayList, and initialized to the empty list. This provides a place to store the individuals whose .Net type is compatible with the declared range of p. A second field, **O.C.myps_nost**, is used to store other values for the range. Of course, if there is no declared range, or the range is Thing, the latter variable will always hold the empty list.

Let $\mathcal{P}'$(C) be the set of ObjectProperties in O whose global domain is declared to be C itself. For each p in $\mathcal{P}'$(C), **O.IC** is given two methods for adding a new range value for p. In one, **Addp**, the parameter type is the interface type corresponding to the range of p. In the other, **Addp_nost**, the parameter type is Thing. The implementations of the former simply add the parameter to **myps**. The implementations for the latter perform a run-time type test and add the parameter to **myps** if the reference has the right type, and to **myps_nost** if not.

[11] If p has multiple global domain declarations, then its domain is their intersection. We invent a name for this class and proceed as if the intersection class had been declared explicitly in the ontology and used as the domain of p.

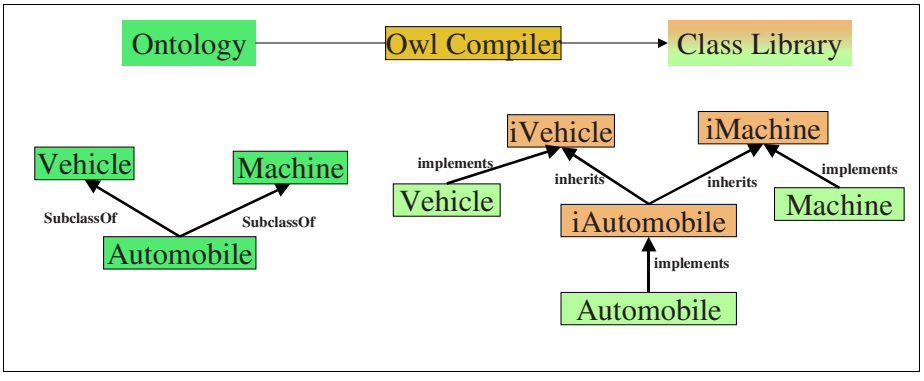

Fig. 4. For each ontology class, the generated library contains both a *class* and an *interface*. *SubclassOf* relationships between ontology classes are mirrored by *inherits* relationships among interfaces. Each library class *implements* the interface with which it is paired.

If p has a global maximum range cardinality of 1, then an additional pair of writing methods is included in **O.IC**. These (**Setp, Setp_nost)** provide a value that becomes the *sole* value in **myps** or **myps_nost.**

For reading, both statically typed and untyped iteration are provided. The statically typed method enumerates values from **myps**, but raises an exception if **myps_nost** is non-empty. The untyped iteration enumerates references from both lists.

If p has a global maxcardinality of 1, an additional statically typed reading method is provided. This method is prototyped as

```
GetP(byval ifnone as O.IR)as O.IR,
```

where R is global range restriction for p. The implementation consists of the following logic:

1. If **mypsnost** is nonempty, an exception is raised. Otherwise
2. If **myps** holds exactly one reference, that reference is returned. Otherwise
3. If **myps** holds no values, the "ifnone" parameter is returned. Otherwise
4. If **myps** holds multiple values, an exception is raised.

If p also has a global mincardinality of 1, an additional method is prototyped as

```
GetP()as O.IR
```

and functions identically to the other **GetP** method, except in case (3), where the second method raises an exception. An implementer's choice among these various methods reflects a decision on whether to treat inconsistency or incompleteness of a model as an *exceptional* situation or as a *normal* circumstance.

The mapping for Datatype properties is completely analogous. Because the fields that store range values always contain lists, regardless of any cardinality restrictions, the absence of a Null value for these types does not pose a problem.

What Is New and What New Is Not

We have just seen the how the class library increments a model with a single triple involving a property from O. But there are a few other kinds of triples that get added to Jena models, and we should look at their counterparts as well. Most interesting is the analog of the triple Jena uses to say "individual I belongs to class C." For Jena,

there is nothing special about the ObjectProperty for "Type". But a statically typed language will somewhere have to treat this as other than just another property. To understand why there is a problem, we revisit the **New** operator mentioned earlier in conjunction with the Ob-O notion of identity. The New operator does more than give a new object an identity. It also gives the object a classification, which must be a single class (not interface) type C. The object returned by this operator is known to the Ob-O runtime to be an instance of that class, to be an instance of the class from which C inherits, etc, back to the built-in class **Object**, which is the root of all class inheritance. Furthermore, the runtime knows that if the object is an instance of any class type C, it is also an instance of the type I for every interface I implemented by C.

But while the **New** operator does a lot of classification, it will not do "additional" classification or "reclassification" or "specialization". If s is a variable bound to a Swimmer, we cannot write s.New.Bowler() to make s be a Bowler as well, or write s.New.NavySeal() to make s a particularly good swimmer. **New** just isn't used like a method; it is a syntactic primitive. So how do Ob-O languages perform these model increments, so straightforward in Jena and its ilk? As pointed out at the end of section 0, they *don't*. A tradeoff has been made in the runtime of these languages to favor performance over this flexibility. Our ontology compiler, in order to regain that flexibility, sacrifices performance.

For any object returned by **Model.NewC()**, we allow applications to create "proxies" of other types at any time. For example, having executed

```
ISwimmer s = m.NewSwimmer()
```

we can later execute

```
IBowler b = m.AsBowler(s)
```

The object created and stored in b is created when AsBowler uses m.NewBowler. The Bowler instance gets added to the **O.Model.my_Bowlers** list. We call the Bowler a *proxy* for the Swimmer instance, and we call the Swimmer a *principal* object of m and *the principal* object for the bowler instance. This information is recorded in private fields of the two instances. We also say that the Swimmer is its own principal object. Either the swimmer or the bowler may be used as the parameter of `m.AsSumoWrestler` to create another proxy. A principal object may end up with proxies of many other types, but at most one per type. If s had already had a proxy of the exact class bowler when we executed `m.AsBowler(s)`, the existing proxy would have been returned. Furthermore, a principal may not have a proxy of a class that is the same as or a superclass of the principal's own class. Having *created* s as a Swimmer, attempting to classify s as an athlete would have no affect on the model -- `m.AsAthlete(s)` would just return the Swimmer s.

The previous paragraph depicts a mechanism that in principal *stores* multiple classifications for a reference. We must also provide alternatives for the .Net runtime's operations that we wish were aware of this. In particular, applications that need proxies also need

- an equality test that treats all objects having the same principal as equivalent,
- a test of whether an object x is an instance of a type T that returns true if any object y having the same principal as x is an instance of T in the .Net sense, and
- a casting operator that will successfully cast x to T if any object having the same principal as x could be cast to T by a .Net cast.

The first method is a straightforward overloading of *Equals*. Since types are first-class objects in the .Net reflection API, the latter two are likewise easy to implement.

Note that this is *not* a matter of implementing inferences authorized by a model and ontology. The swimmer and bowler did not arise from descriptions with distinct ids. They fail the innate identity test solely because, to satisfy the requirements of static typing, two different invocations of the New operator were needed.

Retaining the URI Mapping

The .Net languages contain a weak but still useful means of extending the reflection API. By decorating the declarations of classes, interfaces, methods, etc. with some optional syntax, the .Net compiler arranges for the runtime to create objects called *custom attributes* whenever the library is loaded and associate these objects with the reflection instance corresponding to the decorated unit. There is very little restriction on what can go into classes used as custom attributes. So generating the code

```
<Resource("http://www.w3.org/2001/vcard-rdf/3.0#VCARD"> _
    Public Interface IVCARD  …    End Interface
```

suffices to make the URI used for the class VCARD available via reflection to any application that loads the generated library.

5 Annotations

Ontologies do not address some issues that a programmer considers when designing an Ob-O class library. These are issues that rely on knowledge (or assumptions) about the applications that will actually be implemented using the library.

The four primary considerations that programmers bring to bear in these considerations are code size, data size, performance, and encapsulation. The latter is particularly important to Ob-O programmers, since it is central to their ability to provide structured explanations of algorithm correctness and security.

The scheme outlined in section 0 produces a library that is larger, slower, and less encapsulated than one that a programmer would likely produce manually. It will, on the other hand, be more flexible – that is, supply the functionality needed for a wider variety of possible applications – than the typical manually produced library. We now consider some *annotations* that can be asserted about the classes and properties of an ontology to control these tradeoffs. The list is meant only to be indicative of what is possible, not an exhaustive enumeration of what would be useful.

Class libraries do not always include a root object like *model* whose instances effectively partition the instances of all other objects. Even when they do, a root can generally enumerate the instances of few, if any, reference types in the library. A programmer can authorize the generator to omit the enumeration method for a type by marking the type as non-enumerable. A non-enumerable type may not have an enumerable supertype.

Ob-O programmers are comfortable with explicitly classifying an individual prior to using that individual in a context that requires (implies) its membership in the class -- they must circumvent the static typing system when this is not the case. Most

programmers would argue that the value of trapping unintentional errors in the static type checker far outweighs the occasional inconvenience of having to provide explicit classifications. For most properties, a programmer would be happy to give up the flexibility of adding assertions about individuals not statically known to belong to the declared domain/range class. In other words, they are willing to allow the global domain and range declarations for a property be treated as static prerequisites for explicit assertions.

The .Net languages (and Java) allow a class not only to implement multiple interfaces, but to inherit the implementation of one other class. This means that it shares the implementation of any interfaces it has in common with that class, including fields. This has no impact on functionality or flexibility, but can result in drastically smaller code. Using annotations to indicate a single-inheritance thread through some of the classes in an ontology would significantly reduce the amount of code duplication needed to implement interfaces in the library's classes.

Many types in .Net programs exist *only* to be used as an interfaces – no class is needed that implements just that one interface. For example, an annotation could be used to indicate that "Vehicle" did not need to be supported as a class with its own implementation, but only as an interface implemented by its subclasses.

Classes in .Net programs can be marked as "MustInherit", meaning that any instance of the class must actually be created by instantiating some class that inherits from it. A class defined in an ontology as the union of other classes would be a likely candidate for this annotation.

Enumerated types in programming languages are generally implemented by assigning, at compile time, a distinct integer value to each enumeration member and using the integers as the run-time representation of the member. In .Net, programmers can choose these integers, and a common programming technique is to choose integers whose binary representation can be effectively used as a bit mask, thus enabling some logical set operations to be performed in constant time. This optimization could be expressed through annotations.

Table 1 summarizes the tradeoffs inherent in each of these annotations. The choice of the "default" flexibility to provide in a generated class library could be one that provided great flexibility and required annotations to give up some of that flexibility. Alternatively, the default could be for a relatively lean but inflexible library, requiring annotations to include the additional flexibility.

Table 1. Annotation Tradeoffs

Annotation	Code size	Data size	Perfor-mance	Encap-sulation
Enumerability	☺	☺	☺	
Domain/range as static requirements	☺	☺		☺
InheritsFrom	☺			
InterfaceOnly	☺			☺
MustInherit				☺
Bitmask		☺	☺	

6 Discussion

The idea of generating an Ob-O class library from an ontology is not new. A commercial product from Ontology Works[10] generates Java class libraries from a proprietary ontology language, apparently oriented toward use for accessing a database whose schema is also generated from the ontology. Mechanisms for dealing with multiple inheritance and inconsistency are not described, however.

A number of concerns for generating a viable class library implementation were omitted entirely from the design sketched in the previous sections. I will touch on several of them briefly here.

The generated code must supply *names* for classes, interfaces, properties, methods, and fields. The names must be legal, must avoid collisions with one another and with names in libraries on which the generated class library depends. The public names should be recognizable by a programmer who is familiar with the ontology being implemented. None of this is a serious problem. In both .Net and OWL ontologies, names consist of a namespace and a local part. Keeping a 1-1 correspondence between namespaces avoids most potential collisions, and relying on the local portion of the names from the ontology as the basis for the local portion of the class library names resolves the latter problem.

The discussion omitted operations that *remove* facts from a model. For the most part, those are just the obvious complements of the operations that add the same facts. The one exception is the classifications done by **New**, which inherently have the same lifetime as the object created. One could attempt to hide those classifications from the operations implemented by the library itself, but they would still be visible to the OO runtime system and so could not be effectively hidden from applications.

Scalability is a serious issue. The design presented here views the ontology to be implemented as a self-contained unit. Clearly we want to be able to reuse (share) the binary implementation of one ontology in the implementation of ontologies that import it. But there don't appear to be any restrictions in OWL on how an ontology that imports another one can use the terms of the imported one. The importing ontology may add new superclasses to the classes in the imported ontology, add new properties using classes of the imported ontology as their domain, or add global range requirements to imported properties. This is all very unlike the way class libraries use and extend one another.

The library generator description in this paper largely ignored *Datatypes and DatatypeProperties*. They are much simpler to deal with than Objects and ObjectProperties for three reasons. First, there no possibility of the equality of two datatype values being uncommitted. Second, there is no possibility of *asserting* the type of a datatype value. Third, there doesn't seem to be any practical need to store a datatype value in the range of a DatatypeProperty when the value is the wrong type – it seems that raising an exception in the case where there is no obvious conversion would be perfectly acceptable. It is also hard to imagine a use for a DatatypeProperty with no range type specified. The remaining problem is matching up the XSD types with the .Net value types, and the correspondence is straightforward for all the commonly used types, with the exception of fact that XSD integers have no size limit. Transparently mapping between an ontology's datatypes and built-in types opens up to the application programmer the large body of predefined methods on those types.

The use of proxies and principals in inadequate, as described, to fully deal with the problem of multiple explicit types for a single individual. Consider the example used earlier. Any property declared with Athlete as its domain would have ended up with explicit storage in both the principal, Swimmer, and in the proxy, Bowler, instances. If IsProfessional were such a property, we would have two values around. Of course we know it is possible to be both a professional basketball player and an amateur at baseball, so the same might be true for swimming and bowling. But this just shows the dangers of thinking you can understand the meaning of a property from its name – clearly the semantics of OWL requires that if IsProfessional is a property of athletes, there is no sense in which multiple values of the property can be assigned to an individual and differentiated based on a subclass when the individual happens to belong to more than one subclass.

References

1. DAML Query Language (DQL) : Abstract Specification, April 2003, Richard Fikes., Patrick Hayes, and Ian Horrocks, editors. URL: http://www.daml.org/2003/04/dql/dql
2. OWL Web Ontology Language Overview. Edited by Deborah L. McGuinness (Knowledge Systems Laboratory, Stanford University) and Frank van Harmelen (Vrije Universiteit, Amsterdam). URL: http://www.w3.org/TR/2003/WD-owl-features-20030331/.
3. The Description Logic Handbook: Theory, Implementation and Applications, Edited by Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, Peter Patel-Schneider, published by Cambridge University Press
4. James Gosling,Bill Joy,Guy Steele, and Gilad Bracha The Java Language Specification, Second Edition, Sun Microsystems, Inc, 2000
5. Don Box, Essential .NET Volume I: The Common Language Runtime,Addison-Wesley, November 2002
6. Tim Berners-Lee, James Hendler, Ora Lassila, The Semantic Web, Scientific American, May 2001
7. Brian McBride, Jena: A Semantic Web Toolkit, IEEE Internet Computing, Novermber, 2002, http://dsonline.computer.org/0211/f/wp6jena.htm
8. Mike Dean and Dave Rager, DAML dotnet API, URL: http://www.daml.org/2002/10/dotnetAPI/
9. Reference description of the DAML+OIL (March 2001) ontology markup language,2001, http://www.daml.org/2001/03/reference.html
10. OWL and the IODE, Ontology Works whitepaper, September 2001, http://www.ontologyworks.com/docs/whitepaper.pdf