14. September 2000

An informal description of OIL-Lite and Standard OIL:

a layered proposal for DAML-O

Sean Bechhofer (1),
Jeen Broekstra (3),
Stefan Decker (4),
Michael Erdmann (5),
Dieter Fensel (2),
Carole Goble (1),
Frank van Harmelen (2,3),
Ian Horrocks (1),
Michel Klein (2),
Deborah McGuinness (4),
Enrico Motta (7),
Peter Patel-Schneider (6)
Steffen Staab (5),
Rudi Studer (5)

(1) Department of Computer Science, University of Manchester, UK,
{horrocks,carole}@cs.man.ac.uk

(2) Vrije Universiteit Amsterdam, Netherlands,
{dieter, frankh, mcaklein}@cs.vu.nl

(3) AIdministrator Nederland B.V., Amersfoort, Netherlands
jeen.broekstra@aidministrator.nl,
frank.van.harmelen@aidministrator.nl

(4) Stanford University, USA
stefan@db.stanford.edu, dlm@ksl.stanford.edu

(5) AIFB, University of Karlsruhe, Germany,
{mer, sst, rst}@aifb.uni-karlsruhe.de

(6) Bell Laboratories, Murray Hill, USA
pfps@research.bell-labs.com

(7) Knowledge Media Institute, The Open University, UK,
e.motta@open.ac.uk

This document provides an informal description of the modeling primitives of the OIL dialects "OIL-Lite" and "Standard OIL". It only gives a compact and informal description of these languages, plus a simple illustrative example. For more discussion and motivation, see the papers at http://www.ontoknowledge.org/oil/papers.shtml.

OIL's machine readable syntax is defined as an XML DTD (http://www.ontoknowledge.org/oil/dtd/), an XML Schema definition (http://www.ontoknowledge.org/oil/xml-schema/) and an RDF Schema definition (http://www.ontoknowledge.org/oil/rdf-schema/). To improve human readability OIL, also has a more compact pseudo syntax where keywords are indicated by **bold faced** text, and grouping of sub-content is indicated by indentation. The formal definition of this human-readable syntax can be found at http://www.ontoknowledge.org/oil/syntax. In this document, we give an informal description of this human-readable syntax, since it is the easiest way to get acquainted with the modelling primitives of the language. We refer to the above URL's for the formal definitions.

An informal description of OIL-Lite

An OIL ontology is a structure made up of several components, some of which may themselves be structures, some of which are optional, and some of which may be repeated. We will write **component**$^{?}$ to indicate an optional component, **component**$^{+}$ to indicate a component that may be repeated one or more times (i.e., that must occur at least once) and **component*** to indicate a component that may be repeated zero or more times (i.e., that may be completely omitted). In general, we assume that identifiers (for classes, roles, etc) do not coincide with keywords of the language (such as **slot-constraint**, **or**, etc). However, this section is intended to be descriptive rather than precise: for a precise formal definition please refer to the URL's mentioned above.

An OIL ontology is delineated by the keywords **begin-ontology** and **end-ontology**, and consists of the actual *ontology definition*, defining a particular ontological vocabulary, preceded by an *ontology container*, which is concerned with describing features of such an ontology, like author, name, subject, etc. For representing metadata of ontologies, we make use of the Dublin Core Metadata Element Set (Version 1.1) [Dublin Core] standard. We will discuss both elements of an ontology specification in OIL. We start with the ontology container and will then discuss the backbone of OIL, the ontology definition.

Ontology Container

We adopt the components as defined by the Dublin Core Metadata Element Set, Version 1.1 for the *ontology container* part of OIL. Although every element in the Dublin Core set is optional and repeatable, in OIL some elements are required or have a predefined value. Required elements are written as **element**$^{+}$. Some of the elements can be specialized with a *qualifier* which refines the meaning of that element. In our shorthand notation we will write **element.qualifier**. The precise syntax based on RDF is given in [Miller et al., 1999], and in the appendix. Here we provide our pseudo-XML syntax explained above.

title^{+} The name of the ontology, e.g., "African animals".

creator^{+} The name of an agent (i.e., a person, a group of persons, or a software agent) that created the ontology.

subject* Keywords or classification code describing the subject the ontology deals with.

description Natural language text describing the content of the ontology, e.g., “A didactic example ontology describing African animals”. Besides this description, there is one special description element required, which has the **release** qualifier:

description.release The version of the ontology (a number), e.g, 1.01.

publisher* Defining the entity that is responsible for making the resource available.

contributor* The name of an agent (i.e., a person, a group of persons, or a software agent) that helped to create the ontology.

date* The date the ontology has been created, modified, or made available (see ISO 8601 for format instructions).

type+ The nature of the resource. A predefined and required value is *ontology*, although this value is not yet in the Working Draft of the resource types [Guenther, 1999].

format* The digital manifestation of the resource, recommended as a value is the MIME type of the resource, i.e. “`text/xml`”.

identifier+ The **URI** of the ontology.

source* Optional references (**URI**) to sources from which the ontology is derived. E.g., a reference to a plain text description of the domain on which the ontology is based.

language+ The language of the ontology. Obviously, one predefined and required value is “`OIL`”. Other elements can contain the language of the content of the ontology, according to RFC 1766.

relation* A list of references to other OIL ontologies. It is recommended to list all ontologies that are imported in the definition section with a **hasPart** qualifier. Other possible and meaningful qualifiers are **replaces**, **isReplacedBy**, **requires** and **isRequiredBy**. For example, to list an imported ontology, we write: **relation.hasPart** “`http://www.ontosRus.com/animals/jungle.onto`”.

rights* Information about rights held in and over the ontology.

Ontology definition

Apart from various header fields encapsulated in its container, an OIL ontology consists of a set of definitions, optionally preceded by an important statement:

import? A list of one or more references to other OIL modules that are to be included in this ontology. Each reference consists of a **URI** specifying where the module is to be imported from, e.g., “`http://www.ontosRus.com/animals/jungle.onto`”. XML schemas and OIL provide the same (limited) means for composing specifications. Specifications can be included and the underlying assumption is that names of different specifications are different (via different prefixes).[1]

definition* Zero or more class definitions (**class-def**), axioms (**disjoint**, **covered**, **disjoint-covered**, **equivalent**), and slot definitions (**slot-def**), the structure of which will be described below.

A class definition **(class-def**) associates a class name with a class description. A **class-def** consists of the following components:

type? The type of definition. This can be either **primitive** or **defined**; if omitted, the type defaults to **primitive**. When a class is **primitive**, its definition (i.e., the combination of the following **subclass-of** and **slot-constraint** components) is taken to be a necessary but not sufficient condition for membership in the class. For example, if the primitive class elephant is defined to be a sub-class of animal with a slot constraint stating that *skin-color* must be grey, then all instances of elephant must necessarily be animals with grey skin, but there may be grey-skinned animals that are not instances of elephant. When a class is **defined**, its definition is taken to be a necessary *and* sufficient condition for membership of the class. For example, if the defined class carnivore is defined to be a sub-class of animal with a slot constraint stating that it *eats* meat, then all instances of carnivore are necessarily meat eating animals, and every meat eating animal is also an instance of carnivore.

name The name of the class (a string).

documentation? Some documentation describing the class (a string).

subclass-of? A list of one or more *class-expressions*, the structure of which will be described below. The class being defined in this **class-def** must be a sub-class of each of the class-expressions in the list.

slot-constraint* Zero or more slot-constraints, a special kind of class-expression, the structure of which will be described below (note that a slot-constraint defines a class). The class being defined in this **class-def** must be a subclass of each slot-constraint.

A class-expression can be either a class name (some of which are *built-in*), an *enumerated-class*, a **slot-constraint**, or a boolean combination of class expressions using the operators **and**, **or** and **not**. The structure of these boolean combinations is as follows (note that they must be parenthesized):

and: A list of two or more class expressions that is to be treated as a conjunction. For example:

(meat **and** fish)

defines the class whose instances are all those individuals that are instances of both the class meat and the class fish.

or: A list of two or more class expressions that is to be treated as a disjunction. For example:

(meat **or** fish)

defines the class whose instances are all those individuals that are instances of either the class meat or the class fish.

not: An expression taking as a parameter a single class expression that is to be negated. For example,

[1] This definition is embryonic.

(**not** meat)

defines the class whose instances are all those individuals that are not instances of the class meat.

Note that class expressions are recursively defined, so that arbitrarily complex expressions can be formed. For example

(**not** (meat **or** fish))

defines the class whose instances are all those individuals that are not an instances of either the class meat or the class fish.

The built-in class names consist of **top**, **thing** and **bottom**. The meaning of these classes is pre-defined. **top** and **thing** are alternative names for the most general class. Every class is a sub-class of **top**, and every individual is an instance of **top**. **bottom** is the name of the least general (sometimes called empty or inconsistent) class. **bottom** is a sub-class of every class, and no individual is an instance of **bottom**.

An enumerated-class is a class that is defined by enumerating its instances. An enumerated-class consists of the key-word **one-of** followed by one or more individual names, with the whole expression being enclosed in parenthesis. For example,

(**one-of** Leo Willie)

defines the class whose instances are Leo and Willie.

In some situations it is possible to use a *concrete-type-expression* instead of a class-expression (e.g., in slot restrictions). A concrete-type-expression defines a range over some data type. Two data types are currently supported: **integer** and **string**. The expression **integer** defines the range of all integers (i.e., -infinity to +infinity) and the expression **string** defines the range of all strings. Sub-ranges can be defined using the expressions (**min** *x*), (**max** *x*), (**greater-than** *x*), (**less-than** *x*), (**equal** *x*) and (**range** *x y*), where both *x* and *y* are either integers or strings. Finally, expressions (of the same type) can be combined using the operators **and**, **or** or **not** as in class expressions. For example,

(**min** 21)

defines the data type consisting of all the integers greater than or equal to 21,

(**less-than** 100)

defines the data type consisting of all the integers less than 100 (i.e., less than or equal to 99),

(**greater-than** “abc”)

defines the data type consisting of all the strings that (lexically) succeed “abc” (e.g., “abd”),

(**or** (**equal** “red”) (**equal** “green”) (**equal** “blue”))

defines the data type consisting of the strings “red”, “green” and “blue”,

(**range** 1 10)

defines the data type consisting of all the integers greater than or equal to 1 and less than or equal to 10 and is equivalent to,

((**min** 1) **and** (**max** 10))

and,

(**equal** “xyz”)

defines the data type consisting of the string “xyz” and is equivalent to,

((**min** “xyz”) **and** (**max** “xyz”)).

A **slot-constraint** is a list of one or more constraints (restrictions) applied to a *slot* (sometimes called a *role* or an *attribute*). A slot is a binary relation (i.e., its instances are pairs of individuals), but a slot-constraint is actually a class definition — its instances are those individuals that satisfy the constraint(s). For example, if the pair (Leo, Willie) is an instance of the slot *eats*, Leo is an instance of the class lion and Willie is an instance of the class wildebeest, then Leo is also an instance of the **has-value** constraint wildebeest applied to the slot *eats*. A **slot-constraint** consists of the following components:

name A slot name (a string). The slot is a binary relation that may or may not be defined in the ontology. If it is not defined, then it is assumed to be a binary relation with no globally applicable constraints, i.e., any pair of individuals could be an instance of the slot.

has-value? A list of one or more expressions (either class-expressions or concrete-type-expressions). Every instance of the class defined by the slot-constraint must be related via the slot relation to an instance of each expression in the list. For example, the **has-value** constraint:

slot-constraint *eats*
has-value zebra wildebeest

defines the class each instance of which *eats* some instance of the class zebra and some instance of the class wildebeest. Note that this does not mean that instances of the slot-constraint eat *only* zebra and wildebeest: they may also be partial to a little gazelle when they can get it. The **has-value** constraint:

slot-constraint *colour*
has-value "red"

defines the class each instance of which has the *colour* "red" (a string). Note that in the absence of some other constraint (e.g., a cardinality constraint), more than one *colour* may be specified. **has-value** *expresses the existential quantifier of Predicate logic*: for each instance of the class, there exists at least one value for this slot that fulfils the range restriction.

value-type? A list of one or more expressions (either class-expressions or concrete-type-expressions). If an instance of the class defined by the slot-constraint is related via the slot relation to some individual x, then x must be an instance or data value of each expression in the list. For example, the **value-type** constraint:

slot-constraint *eats*
value-type meat

defines the class each instance of which *eats* nothing that is not meat. Note that this does not not mean that instances of the slot-constraint eat anything at all. The **value-type** constraint:

slot-constraint *age*
value-type (min 21)

defines the class each instance of which does not have an *age* less than 21. Note that this does not not mean that instances of the slot-constraint have any age at all. **value-type** *expresses the universal (for-all) quantifier of Predicate logic*: for each instance of the class, every value for this slot must fulfill the range restriction.

has-filler? A list of one or more individual names or data values (integers or strings). Every instance of the class defined by the slot-constraint must be related via the slot relation to each individual and data value in the list. For example, the **has-filler** constraint:

slot-constraint *friend*
has-filler Zoe Willie

defines the class each instance of which is a *friend* of both Zoe and Willie. Note that this is equivalent to the **has-value** constraint:

slot-constraint *friend*
has-value (**one-of** Zoe) (**one-of** Willie)

The **has-filler** constraint:

slot-constraint *age*
has-filler 21

defines the class each instance of which has an *age* of 21. Note that this is equivalent to the **has-value** constraint:

slot-constraint *age*
has-value (**equal** 21)

It is also worth reemphasizing that in the absence of other constraints (e.g., *age* being a functional slot), there is nothing to prevent instances of this class having more than one *age*.

max-cardinality? A non-negative integer *n* followed by an expression (either a class-expression or a concrete-type-expression). An instance of the class defined by the slot-constraint can be related to at most *n* distinct instances or data values of the expression via the slot relation. The expression can be omitted, in which case an instance of the class defined by the slot-constraint can be related to at most *n* distinct individuals or data values (regardless of their class or type) via the slot relation. For example, the **max-cardinality** constraint:

slot-constraint *friend*
max-cardinality 2 antelope

defines the class, each instance of which has at most 2 *friends* that are antelopes.

min-cardinality? A non-negative integer *n* followed by an expression (either a class-expression or a concrete-type-expression). An instance of the class defined by the slot-constraint must be related to at least *n* distinct instances or data values of the expression via the slot relation. The expression can be omitted, in which case an instance of the class defined by the slot-constraint must be related to at least *n* distinct individuals or data values (regardless of their class or type) via the slot relation. For example, the **min-cardinality** constraint:

slot-constraint *friend*
min-cardinality 3 wildebeest

defines the class, each instance of which has at least 3 friends that are wildebeests. Note that conflicting cardinality constraints is one way in which logical inconsistencies can arise in an ontology. For example, a class to which both the above min-cardinality and max-cardinality constraints applied would be logically inconsistent (could have no instances) if the ontology correctly represented the fact that a wildebeest is a kind of antelope.

cardinality? A non-negative integer *n* followed (optionally) by an expression (either a **class-expression** or a **concrete-type-expression**). This is simply shorthand for a pair of **min-cardinality** and **min-cardinality** constraints, both with the same *n* and expression. For example,

slot-constraint *friend*
cardinality 1 zebra

is equivalent to

slot-constraint *friend*
max-cardinality 1 zebra
min-cardinality 1 zebra

and defines the class, each instance of which has exactly 1 *friend* that is a zebra.

An axiom asserts some additional fact(s) about the classes in the ontology, for example that the classes carnivore and herbivore are disjoint (can have no instances in common). Valid axioms are:

disjoint A list of two or more class expressions. All of the class expressions in the list are pairwise disjoint, i.e., there can be no individual that is an instance of more than one of the class expressions in the list. For example,

disjoint carnivore herbivore

states that no individual can be an instance of both carnivore and herbivore.

covered A class expression followed by a list of one or more class expressions that cover it. Every instance of the first class expression is also an instance of at least one of the class expressions in the list. For example,

covered animal **by** carnivore herbivore omnivore mammal

states that every instance of Animal is also an instance of at least one of carnivore herbivore omnivore or mammal.

disjoint-covered A class expression followed by a list of one or more class expressions that cover it and that are also pairwise disjoint. Every instance of the first class expression is also an instance of exactly one of the class expressions in the list. For example,

disjoint-covered animal **by** carnivore herbivore omnivore

states that every instance of animal is also an instance of exactly one of carnivore herbivore or omnivore.

equivalent A list of two or more class expressions. All of the class expressions in the list are equivalent (i.e., have the same instances). For example,

equivalent wildebeest gnu

states that an individual is wildebeest if and only if it is a gnu (i.e., wildebeest and gnu are synonyms).

A slot definition (**slot-def**) associates a slot name with a slot description. A slot description specifies global constraints that apply to the slot relation, for example that it is a transitive relation. A **slot-def** consists of the following components:

name The name of the slot (a string).

documentation? Some documentation describing the slot (a string).

subslot-of? A list of one or more **slots**. The slot being defined in this **slot-def** must be a sub-slot of each of the slots in the list. For example,

slot-def *daughter-of*
  subslot-of *child-of*

defines a slot *daughter-of* that is a subslot of *child-of*, i.e., every pair (*x,y*) that is an instance of *daughter-of* must also be an instance of *child-of*.

domain? A list of one or more class-expressions. If the pair (*x,y*) is an instance of the slot relation, then *x* must be an instance of each class-expression in the list. For example,

slot-def *eats*
  domain *animal*

defines a slot *eats* such that any individual that *eats* another individual must be an instance of animal.

range? A list of one or more expressions (either class-expressions or concrete-type-expressions). If the pair (*x,y*) is an instance of the slot relation, then *y* must be an instance or data value of each class-expression or concrete-type-expression in the list. For example,

slot-def *friend*
range *animal*

defines a slot *friend* such that any individual that is a *friend* of another individual must be an instance of animal, and

slot-def *age*
range (**min** 0)

defines a slot *age* such that if the pair (*x,y*) is an instance of *age*, then *y* must be a non-negative **integer**. Note that it is good practice to specify the range data type of a slot that is to be used for data values.

inverse? The name of a slot *S* that is the inverse of the slot being defined. If the pair (*x,y*) is an instance of the slot S, then (*y*,*x*) must be an instance of the slot being defined. For example,

slot-def *eats*
inverse *eaten-by*

defines the inverse of the slot *eats* to be the slot *eaten-by*, i.e., if *x eats y* then *y* is *eaten-by x*.

properties? A list of one or more properties of the slot. Valid properties are:

transitive The slot is transitive, i.e., if both (*x,y*) and (*y*,*z*) are instances of the slot, then (*x,z*) must also be an instance of the slot. For example,

slot-def *bigger-than*
properties *transitive*

defines the slot *bigger-than* to be transitive, so if *Jumbo* the *elephant* is *bigger-than Robbie* the *rhino*, and *Robbie* the *rhino* is *bigger-than Walter* the *warthog*, then *Jumbo* must be *bigger-than Walter*. Note that no slot can be both transitive and functional.

symmetric The slot is symmetric, i.e., if (*x,y*) is an instance of the slot, then (*y*,*x*) must also be an instance of the slot. For example,

slot-def *lives-with*
properties *symmetric*

defines the slot *lives-with* to be symmetric, so if *Zoe* the *zebra lives-with* Willie the *wildebeest*, then *Willie* also *lives-with Zoe*.

functional The slot is functional, i.e., if (*x,y*) is an instance of the slot, then there is no *z* such that (*x*,*z*) is an instance of the slot and *y* is not equal to *z*. For example,

slot-def *has-mother*
properties *functional*

defines the slot *has-mother* to be functional. Note that no slot can be both functional and transitive.

An informal description of Standard OIL

Standard OIL is a strict superset of OIL-Lite. Standard OIL adds to OIL-Lite the possibility to define instances of classes and roles, using the following two constructions:

An **instance-of** statement asserts that an individual is an instance of a class or classes. It consists of an individual name (a string) followed by one or more class-expressions. The individual must be an instance of each of the class-expressions in the list. For example,

instance-of Zoe zebra

states that Zoe is a zebra.

An **related** statement asserts that an individual is related to another individual or data value via a slot relation. It consists of the slot name and an individual name followed by either a second individual name or a data value. The first individual must be related to the second individual or data value via the slot relation. For example,

related *has-mother* Zachariah Zoe

states that Zoe is the *mother* of Zachariah, and

related *age* Zoe 35

states that Zoe is *age* 35.

An example OIL ontology

The following example of an OIL ontology illustrates some of the key features of the language. The ontology is intended purely for didactic purposes and is not to be taken as an example of good modeling practice.

ontology-container
  title "African animals"
  creator "Ian Horrocks"
  subject "animal, food, vegetarians"
  description "A didactic example ontology describing African and Asian animals"
  description.release "1.01"
  publisher "I. Horrocks"
  type "ontology"
  format "pseudo-xml"
  format "pdf"
  identifier "http://www.cs.vu.nl/~dieter/oil/TR/oil.pdf"
  source "http://www.africa.com/nature/animals.html"
  language "OIL"
  language "en-uk"
  relation.hasPart "http://www.ontosRus.com/animals/jungle.onto"

ontology-definitions
    slot-def *eats*
      inverse *is-eaten-by*
    slot-def *has-part*
      inverse *is-part-of*
      properties transitive
    slot-def *comes-from*
    slot-def *age*
      range (min 0)
      properties functional

```
slot-def weight
    range (min 0)
    properties functional
slot-def colour
    range string
    properties functional
class-def animal
class-def plant
disjoint animal plant
class-def tree
    subclass-of plant
class-def branch
    slot-constraint is-part-of
        has-value tree
class-def leaf
    slot-constraint is-part-of
        has-value branch
class-def defined carnivore
    subclass-of animal
    slot-constraint eats
        value-type animal
class-def defined herbivore
    subclass-of animal
    slot-constraint eats
        value-type (plant or (slot-constraint is-part-of has-value plant))
disjoint carnivore herbivore
class-def giraffe
    subclass-of animal
    slot-constraint eats
        value-type leaf
class-def lion
    subclass-of animal
    slot-constraint eats
        value-type herbivore
class-def tasty-plant
    subclass-of plant
    slot-constraint eaten-by
        has-value herbivore carnivore
class-def elephant
    subclass-of animal
    slot-constraint eats
        value-type plant
    slot-constraint colour
        has-filler “grey”
class-def defined adult-elephant
    subclass-of elephant
    slot-constraint age
        has-value (min 20)
```

covered adult-elephant **by** (**slot-constraint** *weight* **has-value** (range 5000 8000))
class-def defined african-elephant
    subclass-of elephant
    slot-constraint *comes-from*
        has-filler Africa
class-def defined indian-elephant
    subclass-of elephant
    slot-constraint *comes-from*
        has-filler India
disjoint-covered elephant **by** african-elephant indian-elephant
class-def kenyan-elephant
    subclass-of elephant
disjoint kenyan-elephant indian-elephant
class-def defined african-animal
    subclass-of animal
    slot-constraint *comes-from*
        has-value ((**one-of** Africa) **or** (**slot-constraint** *is-part-of* **has-filler** Africa))
class-def defined asian-animal
    subclass-of animal
    slot-constraint *comes-from*
        has-value ((**one-of** Asia) **or** (**slot-constraint** *is-part-of* **has-filler** Asia))
class-def defined large-animal
    subclass-of animal
    slot-constraint *weight*
        has-value (min 1000)
class-def continent
class-def country
instance-of Africa continent
instance-of Asia continent
instance-of India country
related *is-part-of* India Asia

Some points to note in the above ontology are:

- The class carnivore is a defined class, and lion can be recognized as a sub-class of carnivore because of its definition.
- The class herbivore is a defined class, and giraffe can be recognized as a sub-class of herbivore because of its definition. However, in this case the inference is a little more complex and is only valid because *has-part* is transitive and *is-part-of* is the inverse of *has-part*.
- The class tasty-plant is inconsistent. This is because tasty-plant is a kind of plant that is eaten by both herbivores and carnivores, but we have already stated that carnivore eat only animals, and that animal and plant are disjoint.
- The class adult-elephant is defined to be all elephants aged 15 (an integer) and over. An adult-elephant is also asserted to have a *weight* between 1500 and 3000 (an integer range). As a result, adult-elephant can be recognized as a sub-class of large-animal.
- The class kenyan-elephant is disjoint from indian-elephant. Moreover, indian-elephant and

african-elephant form a disjoint covering of elephant. As a result, kenyan-elephant can be recognized as a sub-class of african-elephant.

- Africa, Asia and India are individuals, and India is *part-of* Asia. As a result, indian-elephant can be recognized as a sub-class of asian-animal.

Current Limitations of OIL

Our starting point was to define a core language with the intention that additional (and possibly important) features be defined as a set of extensions (still with clearly defined semantics). Modelers will be free to use these language extensions, but it must be clear that this may compromise reasoning support. This seems to us a better solution than trying to define a single "all things to all men" language like Ontolingua. In this section we briefly discuss a number of features which are available in other ontology modeling languages and which are not, or not yet, included in OIL. For each of these features we briefly explain why we chose them, and mention future prospects where relevant.

Default reasoning: Although OIL does provide a mechanism for inheriting values from super-classes, such values cannot be overwritten. As a result, such values cannot be used for the purpose of modeling default values. If an attempt is made at "overwriting" an inherited attribute value, this will simply result in inconsistent class definitions which have an empty extension. For example, if we define the class "CS professor" with attribute "gender" and value "male", and we subsequently define a subclass for which we define the gender attribute as "female", this subclass will be inconsistent and have an empty extension (assuming that "male" and "female" are disjoint).

Rules/Axioms: As discussed above, only a fixed number of algebraic properties of slots can be expressed in OIL, plus a number of axioms relating classes (disjoint covers etc.). These axioms are sufficient for describing arbitrary sububsumption relations (rules) or properties that must hold for all the items in the ontology. For example,

covered Class-expression1 **by** Class-expression2

states that every instance of Class-expression1 is also an instance of Class-expression2, while

covered thing by Class-expression

states that the properties defined in Class-expression must hold for every instance of every class in the ontology. Such a powerful feature is undoubtedly useful, e.g., to enforce the correct interpretation of concepts across multiple integrated ontologies. However, there may also be a requirement for other kinds of rules/axioms with different semantic interpretations.

Further relation properties: The properties that can be specified for relations in OIL are currently restricted to *inverse*, *functional*, *transitivity* and *symmetry*. Other reasonable candidates are *reflexivity*, *irreflexivity*, *antisymmetry*, *asymmetry*, *linearity* (aRb bRa for any pair a,b), *connectivity* (aRb or a=b or bRa for any pair a,b), *partial order* and *total order*. (Note that some of these can be defined in terms of each other). It would also be useful to be able to define *composite* relations.

Modules: OIL contains a very simple construction to modularize ontologies. In fact, this mechanism is identical to the namespace mechanism in XML and XML schema. It amounts to a textual inclusion of the imported module, where name-clashes are avoided by prefixing every imported symbol with a unique prefix indicating its original location. However, much more elaborate mechanisms would be required for the structured representation of large ontologies. Means of renaming, restructuring, and redefining imported ontologies must be available. Future extensions will cover parameterized modules, signature mappings between modules, and restricted export interfaces for modules.

Limited Second-order expressivity: Many existing languages for ontologies (KIF, CycL, Ontolingua) include some form of reification mechanism, which allows the treatment of statements of the language as objects in their own right, thereby making it possible to express statements over these statements. A full second order extension would be clearly undesirable (even unification is undecidable in full 2nd order logic). However, much weaker second order constructions already provide much if not all of the required expressivity without causing any computational problems (in effect, they are simply 2nd order syntactic sugar for what are essentially first order constructions). A precise characterization of such expressivity is required in a future extension of OIL. OIL is currently very restricted. Only classes are provided, not meta-classes or individuals.