State of the Art Report

1. Ontology Representation & Reasoning

Main authors:

Maurizio Lenzerini, Diego Milano, Antonella Poggi

Dipartimento di Informatica e Sistemistica Antonio Ruberti

Universit di Roma La Sapienza ,

Roma, Italy

With contributions from:

- Enrico Franconi, Faculty of Computer Science, Free University of Bozen, Bolzano.
- Daniela Berardi, Dipartimento di Informatica e Sistemistica Antonio Ruberti , Universit di Roma La Sapienza .
- Federica Schiappelli, IASI-CNR, Roma.
- NTNU

1. INTRODUCTION

It is widely accepted to consider an ontology as a conceptualization of a domain of interest, that can be used in several ways to model, analyze and reason upon the domain. Obviously, any conceptualization of a certain domain has to be represented in terms of a well-defined language, and, once such a representation is available, there ought to be well-founded methods for reasoning upon it, i.e., for analyzing the representation, and drawing interesting conclusions about it. This paper surveys both the languages proposed for representing ontologies, and the reasoning methods associated to such languages.

When talking about representation languages, it is useful distinguish between different abstraction levels used to structure the representation itself. In this paper, we essentially refer to three levels, called extensional, intensional, and meta, respectively.

1. The extensional level: this is level where the basic objects of the domain of interest are described, together with their relevant properties.
2. The intensional level: this is the level where objects are grouped together to form concepts, and where concepts and their properties are specified.
3. The meta-level: this is the level where concepts singled out in the intensional level are abstracted, and new, higher level concepts are specified and described, in such a way that the concepts of the previous level are seen as instances of these new concepts.

In this survey, we mainly concentrate on the first two levels, since our primary goal is to provide an account for the basic mechanisms for representing ontologies and for reasoning about them. Categories used in the meta-level of ontologies are very much related to specific applications where ontology languages are used. Therefore, relevant issues concerning the meta-level will be addressed in other surveys of workpackage WP8, in particular those concerning the use of ontologies in interoperability.

This paper is organized as follows. In Section 2 we set up a framework for surveying ontology representation languages. The framework will be then used in Section 3 in the description of specific languages and formalisms, so to have a basic unified view for comparison. Section 4 is devoted to a discussion on reasoning over ontologies.

2. REPRESENTATION

In this section we set up a framework for surveying ontology representation languages. The framework will be then used in Section 3 in the description of specific languages and formalism, so to have a basic unified view for comparison.

The framework is based on three main classification criteria we will use in presenting languages and formalisms. These classification criteria are concerned with the following three aspects:

1. ***What to express***. The first criterion takes into account that ontology is a generic term for denoting domain representation, but specific ontology languages may concentrate on representing certain aspects of the domain. In this paper, we concentrate our attention on classes of languages whose primary goal is to focus on:
   - *Class/relation*. We use this class for referring to languages aiming at representing objects/classes/relations.
   - *Action/process*. We use this class for referring to languages that provide specialized representation structures for describing dynamic characteristics of the domain, such as actions, processes, and workflows. These languages may also incorporate mechanisms for the representation of the static aspects of the domain (e.g., objects and classes), but they usually provide only elementary mechanisms for this purpose, whereas they are much more sophisticated in the representation of the dynamic aspects.
   - *Everything*. We use this class for referring to languages that do not make any specific choice in the aspects of the domain to represent, and, therefore, may be in principle used for any kind of contexts and applications.
2. ***How to express***. This criterion takes into account the basic formal nature of the ontology languages. Under this criterion, we will consider the following classes of languages:
   - Programming languages
   - Conceptual and semantic database models
   - Information system/software formalisms
   - Logic-based
   - Frame-based
   - Graph-based
   - XML-related
   - Visual languages
   - Temporal languages
3. ***How to interpret the expression***. This criterion takes into account the degree in which the various languages deal with the representation of incomplete information iin the description of the domain. Under this criterion, we will consider the following classes of languages:
   - *Single model*. Languages of this class represent a domain without any possibility of representing incomplete information. An ontology expressed in this language should be

interpreted as an exact description of the domain, and not as a description of what we know about the domain. In terms of logic, this means that the ontology should be interpret in such a way that only one model of the corresponding logical theory is a good interpretation of the formal description. This assumption is at the basis of simple ontology languages: for example, if we view the relational model as an ontology languages (where concepts of the domain are represented as relations), then this language is surely a single model -ontology language according to our classification.

- *Multiple models (or, several models)*. Languages of this class represent a domain taking advantage of the possibility of representing incomplete information. An ontology expressed in this kind of language should be interpreted as specifying what we know about the domain, with the proviso that the amount of knowledge we have about the domain can be limited. This point of view is at the basis of sophisticated ontology languages: for example, languages based on First-order logic represent a domain as a logical theory. It might be the case that this theory allows for different legal interpretations, and such interpretations correspond exactly to several models. Thus, languages based on First-order logic are classified as several models -ontology languages.

3. ONTOLOGY LANGUAGES

In this section we analyze different languages and formalisms proposed in the last years for expressing and specifying ontologies. As we said in the introduction, we refer only to languages for the specification of the intensional and the extensional levels. We present the various languages by grouping them according the first classification criterion mentioned in the previous section, namely the one regarding how to express . We refer to several classes of languages, and within a single class, we refer to various languages, each one described in a specific subsection. For each language described in a subsection, we explicitly mention the classes to which the language belongs, according to the three classification criteria of Section 2, namely what to express , how to express , and how to interpret .

3.1 Programming Languages

Programming languages allow representation and manipulation of data in several ways and according to various paradigms. During the years, there has been an evolution towards approaches that favor an abstraction on data types, leading to a cleaner separation between data structures and algorithms that handle them. In the last decade a data-centric approach, the Object Oriented paradigm, has emerged as one of the most influential philosophies, spanning from the programming languages domain to other domains such as that of databases and knowledge representation. The oo approach is itself a loosely defined terms, but is generally associated with a number of concepts, such as complex objects, object identity, methods, encapsulation, typing and inheritance. OO programming languages, such as Java, provide a sophisticated framework for modeling taxonomies of classes and manipulating their instances.

Lately, there have been some attempts to use the Java language as an ontology representation language. In particular, there have been some efforts at mapping ontology languages to Java. Research in this field has been performed mostly in the area of Intelligent Agent systems, as a Java APIs generated from ontologies helps fast implementation of applications or agents consistent with the specification defined in the Ontology itself. For instance, [31] presents OntoJava, a cross compiler that translates ontologies written with Prot g and Rules written in RuleML into a unified java object database and rule engine. In [67], a tool to map OWL ontology into java is presented. Each OWL class is mapped to a Java interface. Several other tools exist or are in the course of being developed for mapping ontology languages to and from java, like java2owl, FRODO rdf2java, sofa etc.

If, despite their Object-orientedness, the use of imperative languages as ontology representation languages may sound strange, this is not the case for logic-programming and functional languages.

Languages like Prolog and LISP have always been used inside knowledge representation systems. There are some recent examples of the use of both Prolog and LISP as ontology representation languages. In particular, a syntax based on a mix of first order logic and LISP is at the base of various ontology representation languages

like Ontolingua and OCML. The Ontolingua Framework has the ability to translate ontologies into Prolog syntax, to allow importing the contents of Ontolingua ontologies into Prolog-based representation systems. XSB inc.[124] offers a suite of ontology tools based on extensional CDF, a formalism in which ontologies can be defined using Prolog-style facts.

After the emerging of the OO paradigm, several attempts have been made to bring this philosophy into functional and even logic-programming languages. An example is F-logic, which many consider particularly suited to represent ontologies and is described in greater detail in the next section. F-logic is used for instance as an internal representation model in the OntoEdit tool[94].

3.1.1 F-logic

WHAT to express:	Everything
HOW to express:	Programming languages
HOW to interpret the expression:	Single model

F-logic or Frame logic [74] is a logical formalism that tries to capture the features of object-oriented approaches to computation and data representation.

It has been developed in order to bridge the gap between object-oriented computing and data representation systems and deductive databases, providing theoretical bases for an object-oriented logic programming language to be used either as a computing tool and as a data representation tool.

Though F-logic is mainly aimed at the representation of object-oriented databases, it is also suitable for representation of knowledge in a frame-based fashion, as frame-based KRS share with object-oriented representation formalisms the central role of complex objects, inheritance and deduction. The name of the language itself is due to this connection with Frame-based languages.

F-logical syntactical framework is not too much dissimilar to that of FOL. As in FOL, the most basic constructs of the language are terms, composed of function symbols, constants and variables. Formulas are constructed from terms with a variety of connectives.

Anyway, elements of the language are intended to represent properties of objects and relations about objects. Terms play the role of logical object ids. They are used to identify objects and access their properties. Function symbols play the role of object constructors. Formulas allow building assertions over objects. In particular, the simplest kind of formulas, called F-molecules, allow asserting either a set of properties for an object, a class-membership relation between two objects or a superclass-subclass relationship between two objects. For example, if O is a term, the syntax O:E represents the fact that O is an instance of E, O::E means that O is a superclass of E and the formula

O[semicolon separated list of method expressions]

Allows to associate various properties to the object denoted by O. Method expressions allow specifying properties that correspond, in an object oriented view, to attributes, methods or signatures. Methods consist of a name, a set of arguments and a return value, which may be scalar or set-valued. Attributes are just methods that don t take any argument. Attributes and method may be specified to be inheritable. Inheritable methods assertions on an object are propagated to objects that are in is-a relationship with the first one, though with a difference between the two kinds of isa: inheritable expressions become not inheritable in member objects. A signature expression specifies a type constraint on the arguments and results of a method. Signatures are used for type checking and are enforced, that is specifying a method without declaring a signature for it raises a type error.

Notice that everything in F-logic is built around the concept of object. There is no distinction between classes and individual objects: they both belong to the same domain. In its most basic version, F-logic doesn t make any distinction between complex objects and atomic values, either.

F-logic is a logic with model-theoretic semantics. Semantics for the F-language are given in terms of F-structures and satisfaction of F-formulas by F-structures, and a notion of logical entailment is given. In [74], a resolution-based proof theory for F-logic is provided. A deductive system consisting of twelve inference rules and one axiom is provided and it s proved to be sound and complete.

Anyway, F-logic s intended use is as a logic programming language, to be used both as a computational formalism and as a data specification language. [74] describes Horn F-programs, which are made of the equivalent of Horn rules in F-logic, and generalized Horn F-programs. As in classic logic programming, the semantics of Horn F-programs is based on the concept of least models. For a class of generalized programs, a unique canonic model may also be found, in a way similar to that used to define semantics of locally stratified programs in classic logic programming.

Note that the syntactic rules defined for F-logic and its deductive system cannot really enforce that in type-constraint stated with signature rules hold in a canonic model of an F-program. In [74], an additional semantic formalization of type rules is given in order to justify the use of static type checking.

Frame logic s syntax has some higher-order features. Despite this higher order syntax, the underlying semantic formally remains first order, which allows avoiding difficulties normally associated with higher order theories.

3.2 Frame-based languages

Frame based languages are based on the notion of frame , first introduced by Minsky to explain certain mental activities[87]. Departing from the initial, largely comprehensive meaning that Minsky gave to this term, the concepts of frame and frame-systems have evolved over time assuming a specific meaning in the context of Knowledge Representation[39][69].

A frame is a data structure that provides a representation of an object or a class of objects or a general concept or predicate. Whereas some systems define only a single type of frame, other systems distinguish two or more types, such as class frames and instance frames.

Frames have components called slots. The slots of a frame describe attributes or properties of the thing represented by that frame and can also describe binary relations between that frame and another frame. In the simplest model of a slot, it has a name and a value. In addition to storing values, slots often also contain restrictions on their allowable values, in terms of data types and/or ranges of allowable values. They may also have other components in addition to the slot name, value and value restrictions, for instance the name of a procedure than can be used to compute the value of the slot. These different components of a slot are usually called its facets. Implemented systems often add other properties to slots, or distinguish between different slot types, as in the case of frames.

The description of an object type can contain a prototype description of individual objects of that type; these prototypes can be used to create a default description of an object when its type becomes known in the model.

In typical frame-based systems, constructs are available that allow to organize frames that represent classes into taxonomies. In a taxonomic hierarchy, each frame is linked to one or, in some systems, more than one parent frame. Through taxonomic relations, classes may be described as specializations of other more generic classes. Classes in a taxonomy inherit from their super classes features like slot definitions and default values. The precise way in which inheritance is implemented may differ from system to system.

One of the characteristics of frame systems is that information at the top of a class hierarchy is fixed, and as such can provide expectations about specific occurrences of values at individual frames. No slot values are ever left empty within a frame; rather they are filled with "default" values, which are inherited from their ancestors. These default values form the stereotypical object, and are overwritten by values that better fit the more specific case.

When extracting values from slots, a system may first look if a specific value is already specified for a slot. If it is, then this can be returned as it is assumed to be the most reliable and up-to-date value for that slot. Failing this, the system can either look at successive ancestors of the frame in question and attempt to inherit values which have been asserted into one of these frames (which is known as "value inheritance") or the system can look at the frame itself and its ancestors for procedures (known as "daemons") which specify how to calculate the required value (which is known as "if-needed inheritance"). If none of the above succeeds in obtaining a value, then the system resorts to extracting the default values stored for the prototype object in question.

3.2.1 Ontolingua

WHAT to express:	Class/relation
HOW to express:	Frame Based
HOW to interpret the expression:	Several models

The name Ontolingua is usually employed to denote both a System for the management of portable ontology specifications and the language therein used. The Ontolingua tool, first introduced in a work by Gruber[48][49][36], is not a knowledge representation system. It is a tool to translate ontologies from a common, shared language to the languages or specification formalisms used in various knowledge representation systems.

The idea behind it is that to allow sharing of ontologies among different groups of users, adopting different representations, it is better to have them first represented in a single, expressive language with clearly defined semantics. Ontologies described in such language may be then translated into implemented systems, suitable for the task at hand, possibly with limited expressive and reasoning capabilities. In order to give clear, declarative semantics to ontology specifications, the Ontolingua syntax and semantics are built on top of those of KIF, the knowledge interchange format (cf. Section 3.6.1.1). Internally, an Ontolingua ontology is a represented with a theory of KIF axioms.

Anyway, to express ontologies through the fine-grained constructs available in the predicate calculus is not a straight task. Moreover, this is also far from the spirit of many knowledge representation tools, that often adopt an object-oriented or Frame based approach to representation. Allowing users to specify ontologies in a familiar frame-like style helps both the ontology designer and the developer of a translation module. To meet this purpose, Ontolingua offers a definition language that tries to abstract from specific representation languages offered by various systems and to offer a description facility which is close in spirit to many Frame-based and Object Oriented knowledge representation languages. Ontolingua definitional forms act as wrappers around sets of KIF sentences, allowing better handling and pushing the user to use just certain easily translatable constructs.

Two main mechanisms support the Frame-oriented approach in handling KIF constructs. First, Ontolingua features a simple definition language that allows to associate a symbolic name to a textual string (for documentation), a set of variables and a set of KIF axioms, through LISP-style forms. Forms are provided to define this way relations, classes, functions, objects and theories (the name theory denote sets of Ontolingua definitions in the Ontolingua terminology). Any legal KIF sentence may be used in these definitions. At a logical level, Ontolingua definitions are equivalent to set of KIF sentences (and they may be easily translated into their KIF equivalents, which are used for internal treatment). For example, using the define-class construct is equivalent to specifying a set of assertions on a unary-predicate symbol. Definitional forms serve as a mean to guide the developer in specifying axioms consistently with a conceptualization of modeling construct, which is inherently frame-oriented. Imposing a structure on the definitions used in the ontologies helps the translation task and guides a user in specifying portable ontologies.

Second, an ontology (an Ontolingua theory) called the Frame-ontology specifies a set of terms trying to capture conventions used in many frame systems. The Frame ontology specifies only second-order relations, defining them through KIF axioms. It serves to diverse purposes. It associates precise semantics to concepts used for ontology modeling in Ontoligua itself. For example, a relation in Ontolingua is defined as a set of tuples, based on the set and tuple concepts which are built-in into KIF. A Class in Ontolingua is defined as a unary relation. The axioms contained in the Frame-ontology are implicitly added to each ontology defined through Ontolingua. One can think of KIF plus the Frame ontology as a specialized representation language. Second order terms defined in the Frame Ontology may be used during modeling (for example, in the context of definition forms) as a compact representation for their first order equivalents. The canonical form produced by Ontolingua during translations is based on these second order relations. Ontolingua is capable of recognizing different variants of expanded forms of these relations and converting them to their compact form. This helps overcome stylistic differences among different ontologies. The terms contained in the Frame Ontology are the only second-order constructs for which Ontolingua guarantees translation to the target implemented systems. Defining these constructs in a separate way allows to improve translation, as they are usually linked to special, ad hoc constructs found in the target languages. To help developers, various definition styles and syntactic variants are allowed. Ontologies defined with this language are then normalized translating them into a single, canonical form and passed to different modules that perform the task of translating them to specific system s representation languages or formalisms.

As already noticed, Ontolingua is not a system designed to represent and query knowledge, but a tool to manage portable ontology definitions and to translate them to a set of implemented KR systems. The KIF language, which is used to represent Ontolingua ontologies, has been designed as a mean to communicate knowledge, representing it in a way that is both parsable and human readable, and not to support automated reasoning. It is very expressive, so much that is impossible to implement a tractable theorem-prover, which can answer arbitrary queries in the language. Conversely, most implemented KR systems typically support limited reasoning over a restricted subset of full first order logic. The completeness of the translation of general KIF expressions is limited by the expressiveness of target systems. Only a subset of the legal KIF sentences will be properly handled by a given implemented representation system into which Ontolingua is translating.

As described above, Ontolingua imposes a structure on the definition of terms and is only guaranteed to translate sentences written with a restricted set of easily recognized second-order relations (those contained in the Frame-Ontology). Using the provided definitional forms and second order relations and avoiding certain special constructs found in KIF (such as the non-monotonic consis operator) favors portability. Furthermore, it facilitates the task of translation. However, there is no real limitation on the KIF constructs that may be used in representation. Everything that is legal in KIF 3.0 is also legal in the specification of an Ontolingua theory.

3.2.2 OCML

WHAT to express:	Class/relation
HOW to express:	Frame based
HOW to interpret the expression:	Single model

OCML[89] is a knowledge modeling language originally developed in the context of the VITAL project, which was aimed at providing a framework to support development of knowledge-based systems over their whole life cycle. Its main purpose is to support knowledge level modeling. This implies that it focuses on logical more than on implementation level primitives. This approach is consistent with other approaches to knowledge modeling such as that of Ontolingua (cf. Section 3.2.1). In fact, OCML has been designed with an eye at compatibility with emergent standards such as Ontolingua itself, and its main modeling facilities closely resemble those of Ontolingua.

Differently from Ontolingua, however, OCML is a representation language that is directly aimed at prototyping KB applications. It has operational semantics and provides interactive facilities for theorem proving, constraint checking, function evaluation, evaluation of forward and backward rules. Moreover, in order to allow different approaches to the development of knowledge based applications and to simplify fast-prototyping, OCML provides also non-logical facilities such as procedural attachments. Apart from its operational nature, another difference is that OCML is not only aimed at representing terminological knowledge, but also behavior. This is supported by primitives that allow to specify control structures. While formal semantics may be provided for a subset of OCML constructs, based on that of the corresponding Ontolingua constructs, no formal semantics is provided for the control language and for procedural attachments.

Further differences are due to the fact that OCML has been influenced by the ideas at the base of the Task/method/domain/application (TDMA) framework, that follows a reuse-oriented approach to the development of knowledge-based applications. In order to support this framework, OCML provides primitives to model both domain knowledge and mapping knowledge, which is used to integrate knowledge from generic, reusable knowledge components such as domain independent problem solving methods and multi-functional domain models.

Extensions to the language suitable to specify task and method components are provided as a separated representation ontology. Indeed, OCML comes with a pre-built base-ontology that has a role closely similar to that played by the Frame-Ontology in Ontolingua, except that the definition it contains are operational and it also defines terms needed for task and method modeling.

OCML s syntax, similarly to that of Ontolingua, is defined by a mixture of LISP style definitional forms and KIF syntax (cf. Section 3.6.1.1). It is based on three basic constructs, namely functional terms, control terms and logical expressions. Functional terms and logical expressions may be used to specify relations, functions, classes, slots, instances, rules, and procedures.

Relations, function, classes, slot, procedures and rules definitions allow to specify semantics mixing constructs with formal and operational meaning. Furthermore, some constructs play both-role. For example, in

the context of a relation definition, the iff-def keyword specifies that a necessary and sufficient condition hold. It also implies that the condition may be used to enforce a constraint and that the proof mechanisms may be used to decide whether or not the relation holds for some arguments. Procedures define actions or sequences of actions that cannot be characterized as functions between input and output arguments (for instance, updates on slot values). OCML provides an interactive tell/ask interface. The same keywords used to interact with this shell are also used in procedures definitions to specify changes in a KB.

3.2.3 OKBC/GFP

WHAT to express:	Class/relation
HOW to express:	Frame Based
HOW to interpret the expression:	Single model

OKBC (Open Knowledge Base Connectivity)[20][21] and its predecessor GFP (Generic Frame Protocol)[68] are not languages for the representation of Knowledge Bases. They are APIs and reference implementations that allow to access and interact in a uniform way with knowledge bases stored in different knowledge representation systems. They were developed to allow knowledge application authors to write representation tools (e.g., graphical browsers, frame editors, analysis tools, inference tools) in a system-independent (and thus interoperable) fashion, shielding applications from many of the idiosyncrasies of a specific KRS. GFP was primarily aimed at systems that can be viewed as frame representation systems, while OKBC aims have been extended to general KRSs.

OKBC provides a uniform knowledge model of KRSs, based on a common conceptualization of knowledge bases, classes, individuals, slots, facets, and inheritance. It also specifies a set of operations based on this model (e.g., find a frame matching a name, enumerate the slots of a frame, delete a frame). The OKBC knowledge model is an implicit representation formalism that underlies all the operations provided by OKBC. It serves as an implicit interlingua for knowledge that is being communicated using OKBC, and systems that use OKBC translate knowledge into and out of that interlingua as needed.

The OKBC Knowledge Model supports an object/frame-oriented representation of knowledge and provides a set of representational constructs commonly found in frame-based knowledge representation systems. To provide a precise and succinct description of the OKBC Knowledge Model, the OKBC specification uses KIF, the Knowledge Interchange Format (cf. Section 3.6.1.1) as a formal specification language.

A *frame* is a primitive object that represents an entity in the domain of discourse. A frame has associated with it a set of *own slots*, and each own slot of a frame has associated with it a set of entities called *slot values*. Formally, a slot is a binary relation. An own slot of a frame has associated with it a set of *own facets*, and each own facet of a slot of a frame has associated with it a set of entities called *facet values*. Formally, a facet is a ternary relation. Slot and facets are in the universe of discourse and may optionally be represented by frames.

A *class* is a set of entities and is represented with a unary relation. Each of the entities in a class is said to be an *instance of* the class. An entity can be an instance of multiple classes, which are called its *types*. A class can be an instance of a class. A class that has instances being themselves classes is called a *meta-class*. Entities that are not classes are referred to as *individuals*. A class is represented through a frame. A frame that represents a class is called a *class frame*, and a frame that represents an individual is called an *individual frame*.

A class frame has associated with it a collection of *template slots* that describe own slot values considered to hold for each instance of the class represented by the frame. The values of template slots are said to *inherit* to the subclasses and to the instances of a class. The OKBC knowledge model includes a simple provision for default values for slots and facets. Template slots and template facets have a set of *default values* associated with them. Intuitively, these default values inherit to instances unless the inherited values are logically inconsistent with other assertions in the KB, the values have been removed at the instance, or the default values have been explicitly overridden by other default values.

A *knowledge base* (KB) is a collection of classes, individuals, frames, slots, slot values, facets, facet values, frame-slot associations, and frame-slot-facet associations. KBs are considered to be entities in the universe of discourse and are represented by frames. The frames representing KBs are considered to reside in a distinguished KB called the *meta-kb*, which is accessible to OKBC applications.

The OKBC Knowledge Model includes a collection of classes, facets, and slots with specified names and semantics. It is not required that any of these standard classes, facets, or slots be represented in any given KB, but if they are, they must satisfy the semantics prescribed by the OKBC specification. For example, the OKBC specification defines the meaning of the class :CLASS, which always maps to the class of all classes.

3.2.4 XOL

WHAT to express:	Class/relation
HOW to express:	Frame Based
HOW to interpret the expression:	Single model

The language called XOL[70] was inspired by Ontolingua (cf. Section 3.2.1) and OML. As Ontolingua, it has been designed as an intermediate language for transferring ontologies among different database systems, ontology-development tools, or application programs. Its syntax is based on XML, and its semantics are defined as a subset of the OKBC knowledge model called OKBC-Lite. OKBC-Lite extracts most of the essential features of OKBC, while not including some of its complex aspects.

The design of XOL uses what its authors call a generic approach to defining ontologies, meaning that a single set of XML tags (described by a single XML DTD) defined for XOL can describe any and every ontology. This approach contrasts with the approaches taken by other XML schema languages, in which typically a generic set of tags is used to define the schema portion of the ontology, and the schema itself is used to generate a second set of application-specific tags (and an application-specific DTD) that in turn are used to encode a separate XML file that contains the data portion of the ontology (cf. Section 3.7.1).

The ontology building blocks defined by the OKBC-Lite Knowledge Model include classes, individuals, slots, facets, and knowledge bases. A main difference from the OKBC knowledge model (cf. Section 3.2.3) resides in that all references to the notion of ‘‘frame’’ have been eliminated. Other differences lie inrestrictions to the kind of facets and of basic data types available. OKBC-lite treats classes in much a similar way as OKBC. A *class* is a set of entities. Each of the entities in a class is said to be an *instance of* the class. An entity can be an instance of multiple classes, which are called its *types*. A class can be an instance of another class. A class that has instances that are themselves classes is called a *meta-class*. Entities that are not classes are referred to as *individuals*. Thus, the domain of discourse consists of individuals and classes. In the XOL specification, classes and individuals are generically referred to as entities (but note that entities is just a common name to refer to both classes and individuals and not a concept of the model).

The OKBC-Lite knowledge model assumes that every class and every individual has a unique identifier, also known as its *name*. The name may be a string or an integer, and is not necessarily human readable. As already mentioned, frames are not considered in the OKBC-lite model. Thus, classes and individuals stand for themselves and are not represented through frames. The concept of own slot is transferred to both classes and individuals. Note that, though slots have a unique name like classes and individuals, they may only be used in conjunction with the class that owns them. The same holds for slot values or facets with regards to the owner slot. Instance-of, type—of subclass-of and superclass-of relations may be specified betweenclasses or classes and individuals. Template slots and facets and default values are also modeled in a way similar to how they are modeled in OKBC, and inherited in a pretty similar fashion.

A knowledge base is a collection of classes, individuals, slots, slot values, facets, and facet values. A knowledge base is also known as a module. As the OKBC knowledge model, the OKBC-Lite knowledge model includes a collection of classes, facets, and slots with standard names and semantics.

Differently from the OKBC model, the semantics of the OKBC-lite model are only given in an informal way. The XOL DTD defines the structure of XOL documents, by XML-izing the conceptual components of the OKBC-lite model. The DTD defined in the specification gives necessary but not sufficient rules for the well-formedness of XOL documents. Other rules, for example regarding uniqueness of names and order between elements are specified in the specification in an informal way.

3.3 Conceptual and semantic data models

In the 70s, traditional data models (relational, hierarchical and network), were gaining wide acceptance as bases for efficient data management tools. Data structures used in these traditional models are relatively close to those used for the physical representation of data in computers, ultimately viewing data as collections of records with printable or pointer field values. Semantic (or conceptual) models [62] were introduced primarily as schema design tools: a schema could be first designed in a high level semantic model and then translated into one of the traditional, record oriented models such as the relational one, for ultimate implementation. Examples of proposed semantical data models are The ER and Extended ER data model, FDM (Functional data model), SDM (Semantic Data Model). The emphasis of the initial semantic models was then to accurately model data relationships that arise frequently in typical database applications. Consequently, semantic models are more complex than the relational model and encourage a more navigational view of data relationships.

Semantic models provide more powerful abstractions for the specification of databases. The primary components of semantic models are the explicit representation of objects, attributes of and relationships among objects, type constructors for building complex types, ISA relationships and derived schema components. In typical implementations of semantic data models, abstract objects are referenced using internal identifiers that are not visible to users. A primary reason for this is that objects in a semantic data model may not be uniquely identifiable using printable attributes that are directly associated with them.

Not all the basic features listed above are supported by all models. Even if they are supported, most models impose some kind of restriction on the use of these constructs. Some of the models were first proposed with a limited subset of the above-mentioned features, and were subsequently extended to include some other features. For example, the original ER model proposed by Chen did not allow ISA relationships. The Extended Entity Relationship model, which includes ISAs, has been proposed later and is now widely used.

Despite the similarity of many of their features, semantic data models differ from object-oriented models in that the former tend to encapsulate the structural aspects of objects, whereas the latter describe more behavioral aspects. Object oriented models do not typically embody the rich type constructors of semantic models. Semantic models also show many analogies with models traditionally used in the AI field, like those based on frames. Frame-based systems allow representation of abstract types, which are typically organized in ISA hierarchies. Anyway, frame-based systems usually don t allow explicit type-construction mechanisms such as aggregation and grouping. Furthermore, many frame systems allow slots to hold executable attachments that behave in a way not dissimilar from methods in object-oriented systems.

After the introduction of Semantic data models, there has been increasing interest in using them as the bases for full-fledged database management systems or at least complete front end to existing systems. A major problem in accomplishing this task was to give them give precise semantics and enable reasoning on conceptual schemas. Recent research has focused on the use of Description Logics for this purpose [19].

Conceptual data schemes and ontologies share many similarities, and there are proposals of using conceptual methodologies and tools for ontology modeling. For example, [65] proposes a methodology for ontology building with semantic models and a markup language to exchange conceptual diagrams at run-time. The proposed methodology is applied to the ORM (Object-Role Modeling) semantic model, but it may be extended to other semantic models as well.

3.4 Information systems and software formalisms

Several formalisms to aid design of software artifacts and information systems started to appear since the mid seventies. Between the first to be proposed were the Structured Analysis and Design Technique (SADT) and the Requirements Modeling Language (RML)[45]. Between the mid 1970s and the late 1980s various object-oriented modelling languages and methodologies started to appear. For instance OOA, the Conceptual Modeling Language (CML) [110] and Telos[91]. The number of OO modelling langues continued to increase during the period between 1989-1994, partly because no one single model seemed completely satisfactory to users. By the mid-1990s, new iterations of these methods began to appear and these methods began to incorporate each other s techniques, and a few clearly prominent methods emerged. In late 1994 Grady Booch and Jim Rumbaugh of Rational Software Corporation began their work on unifying the Booch and OMT (Object Modeling Technique) methods. In the Fall of 1995, Ivar Jacobson and his Objectory company joined Rational and this unification effort, merging in the OOSE (Object-Oriented Software Engineering) method. This lead to the deveopment of UML (Unified Modeling Language), which is now one of the most widely used formalisms for Information Systems design.

3.4.1 UML

WHAT to express:	Class/relation
HOW to express:	IS formalism
HOW to interpret the expression:	Single model

UML[115] was originally designed for human-to-human communication of models for building systems in object-oriented programming languages. Over the years, though, its use has been extended to a variety of different aims, including the design of databases schemas, XML document schemas, and knowledge models. Several complementary standards have continued to emerge after the release of UML, such as the OCL (Object Constraint Language) which allows to specify constraints, the OMG MOF[86] (Meta Object Facility).

UML primary drawbacks lie in a lack of formally defined semantics, as the OMG standard gives only an informal specification of UML semantics. Another issue is the lack of a standard textual representation. Anyway, the OMG is in the process of adopting XMI(XML Metadata Interchange)[121] as a standard for stream-based model interchange.

Various attempts have been made to give formal semantics to the language (e.g. [34][24][33]). In particular, latest proposals, such as [18], are based on Description Logics and try to establish a solid basis for allowing automated reasoning techniques, to be applicable to UML class diagrams.

Concurrently, there have been various proposals that advocate the use of UML for ontology modeling and representation. For example,[27] focuses on a subset of the language (In particular the Class Diagram, complemented by the OCL language in order to express constraints and rules). [8] examines similarities and differences between UML and ontology languages like DAML+OIL (cf. Section 3.6.2.3.1), and [52] analyzes the problem of assigning ontologically well-founded semantics to the UML modeling constructs.

3.5 Graph-based models

The ability to display problems and knowledge in a graphical and thus intuitive way has always had an important role in research fields connected to KR. In particular, several formalisms based on various kinds of graph based or graph-oriented notations have been proposed and studied.

In this section we review some of the most influential graph based formalisms. Two of them, namely semantic networks and conceptual graphs, originated from the AI community, and have been subject of research for a long time. OML/CKML is a framework and markup language for knowledge and ontology representation based (also) on conceptual graphs. Conversely, Topic Maps are a quite recent proposal that originated from the web and XML community.

3.5.1 Semantic Networks

WHAT to express:	Class/relation
HOW to express:	Graph based models
HOW to interpret the expression:	Single model

Semantic networks are knowledge representation schemes involving nodes and links (arcs) between nodes. The nodes represent objects or concepts and the links represent relations between nodes. The links are directed and labeled; thus, a semantic network is a directed graph. The term semantic net appeared when Richard H. Richens of the Cambridge Language Research used them in 1956 as an "interlingua" for machine translation of natural languages. The term semantic networks dates back to Ross Quillian's Ph.D. thesis (1968)[99], in which he first introduced it as a way of talking about the organization of human semantic memory, or memory for word concepts. The idea of Semantic Networks, though, that is of a network of associatively linked concepts, is very much older, as Porphyry s tree (3rd century AD), Frege s concept writing (1879), Charles Peirce Existential Graphs (1890 s), Steltz s concept hierarchies (1920 s) and other similar

formalisms used in various fields (philosophy, logic, psychology, linguistics) may all be considered examples of semantic networks.

Quillian's basic assumption was that the meaning of a word could be represented by the set of its verbal associations. From this perspective, concepts have no meaning in isolation, and only exhibit meaning when viewed relative to the other concepts to which they are connected by relational arcs. In Quillian's original semantic networks, a relation between two words might be shown to exist if, in an unguided breadth-first search from each word, there could be found a point of intersection of their respective verbal associations. While Quillian's early nets might have appeared to be an attractive psychological model for the architecture of human semantic knowledge, they did not provide an adequate account of our ability to reason with that knowledge. A couple of years later a psychologist, Allan Collins, conducted a series of experiments along with Quillian to test the psychological plausibility of semantic networks as models. The networks they used gave far greater prominence than before to the hierarchical organization of knowledge, being organized as taxonomic trees or isa hierarchies in which each node stated properties that are true of the node and properties of higher nodes are inherited by the lower nodes to which they are connected, unless there is a property attached to a lower node that explicitly overrides it.

As semantic networks seemed to mimic the way the human mind structures knowledge, they were regarded has an attractive way to represent knowledge in AI Knowledge Representation System.

Semantic networks as a representation of knowledge have been in use in artificial intelligence (AI) research in a number of different areas. Another major area in which the use of semantic networks is prevalent, is in models based on linguistics. These stem in part from the work of Chomsky. This latter work is concerned with the explicit representation of grammatical structures of language. It is opposed to other systems that tried to model, in some machine-implementable fashion, the way human memory works. Another approach combining aspects of both the previously mentioned areas of study was taken by Schank in his conceptual dependency approach. He attempted to break down the surface features of language into a network of deeper, more primitive concepts of meaning that were universal and language independent.

Despite their influence, a major problem with semantic networks was a lack of clear semantics of the various network representations. In particular, they lacked a clear meaning for what nodes and arcs represented. Semantic networks do tend to rely upon the procedures that manipulate them. For example, the system is limited by the user's understanding of the meanings of the links in a semantic network. No fundamental distinction is made between different types of links, for example those that describe properties of objects, and those that make assertions, nor between classes of objects and individuals. The interpretation of nodes suffers an analogous ambiguity. One interpretation of generic nodes (ones that refer to a general class or concept) is as sets. Thus the relation holding between a node at one level of a hierarchy and that at a higher level is that of subset to superset. If the generic node is interpreted as a set, then the relation of nodes representing individuals to generic nodes in the network is that of set membership. A second interpretation of generic nodes is as prototypes, and of instance nodes as individuals for whom the generic nodes provide a prototypic description. Individuals will then be more or less like the prototype.

Several studies tried to give more precise semantics to networks (e.g. [118][12][1][13]). The research trend originating from these first attempts has lead to the development of Description Logics.

In the 1960s to 1980s the idea of a semantic link was developed within hypertext systems as the most basic unit, or edge, in a semantic network. These ideas were extremely influential, and there have been many attempts to add typed link semantics to HTML and XML.

3.5.2 Conceptual Graphs

WHAT to express:	Class/relation
HOW to express:	Graph based
HOW to interpret the expression:	Single model

A particular use of semantic networks is to represent logical descriptions. Conceptual writing of Frege and the Existential Graphs of Charles S. Peirce represent an early attempt at finding form of diagrammatic reasoning. Conceptual Graphs are a kind of semantic network related to the Existential Graphs of Peirce. They were introduced by John F. Sowa [108].

Their intended aim is to describe concepts and their relations in a way that is both close to human language and formal, in order to provide a readable, but formal design and specification language. They should

also allow a form of reasoning that may be performed directly on their Graphic representation, by means of subsequent transformations applied to the (conceptual) Graph.

Basically, a conceptual graph is a bipartite graph containing two kind of nodes, called respectively concepts and conceptual relations. Arcs link concepts to conceptual relations, and each arc is said to belong to a conceptual relation. There are no arcs that link concepts to concepts or relations to relations. Concepts have an associated type and a *referent. A referent* is a way to denote the entity of the Universe of Discourse to which the concept refers. It consists of a *quantifier*, a *designator*, which either points to the referent or describes it, and a *descriptor*, which is a Conceptual Graph describing some aspects of the referent. Note that a quantifier here may only be of existential kind or specify that a precise number of instances of the referent exist. A descriptor is considered as a Conceptual Graph nested into a concept. The concept is said to be a Context for the nested CG. Conceptual relations are typed, too. A relation type associates to a conceptual relation a valence (equal to the number of arcs that belong to the relation) and a signature that constraint the types of concepts linked to the relation. A complete presentation of CGs is outside of the scope of this document. Some basic notions may be found in [22].

The book published by Sowa received conflicting initial reactions[23][107]. Subsequent papers clearly shown some errors in Sowa s theses, and highlighted the need for a much cleaner formalization of CGs [117]. Various other attempts at formalizing CGs have been proposed during the years. Another problem related to Conceptual Graphs was that, since they have shown to have the same expressive power of first order logic, most of the problems that are interesting when reasoning on conceptual graphs (like validity and subsumption) are undecidable. Several problems remain NP-complete even when reasoning on restricted versions of the Graphs, like Simple Graphs (corresponding to the conjunctive, positive and existential fragment of FOL). To overcome this inconvenient, there have been attempts to identify guarded fragments of CGs[5].

Despite all this problems, the ideas behind Conceptual Graphs seem to have been highly influential. For example, there are many similarities between CGs and the Resource Description Framework, RDF (some similarities and differences are explained by Tim Berners Lee in [11]). Refer to Section 3.7.2.1 for an introduction of RDF.

Sowa is preparing a proposal for a standardization of Conceptual Graphs, to be submitted to ISO[25]. The draft presents conceptual graphs as a graphic representation for logic, with the full expressive power of first-order logic and with extensions to support metalanguage, modules, and namespaces. It describes the abstract syntax on which conceptual graphs are based and also three concrete syntaxes that may be use for actual representation of Conceptual graphs. They are, respectively, a Display Form, which is essentially graphical, a linear form, which is textual, and a machine-readable form, called CGIF (Conceptual Graphs Interchange Format). The draft asserts that CGIF sentences may be converted to logically equivalent sentences in the KIF language, (cf. Section 3.6.1.1). This allows to give CGIF sentences the same model theoretic semantics defined for the KIF language. Anyway, the document is still unclear about the translation rules to be used.

The draft also gives a deductive system for Conceptual Graphs based on six canonical formation rules . Each rule performs one basic graph operation. Logically, each rule has one of three possible effects: it makes a CG more specialized, it makes a CG more generalized, or it changes the shape of a CG while leaving it logically equivalent to the original. All the rules come in pairs: for each specialization rule, there is a corresponding generalization rule; and for each equivalence rule, there is another equivalence rule that transforms a CG to its original shape. These rules are essentially graphic. Each rule is said to have a correspondent inference rule in predicate calculus. These inference rules, however, are not currently shown in the draft.

3.5.3 OML/CKML

WHAT to express:	Everything
HOW to express:	Graph Based
HOW to interpret the expression:	Single model

Conceptual Knowledge Markup Language (CKML)[72][71][92] is an application of XML. It follows the Conceptual Knowledge Processing (CKP) approach to knowledge representation and data analysis and also incorporates various principles, insights and techniques from Information Flow (IF), the logical design of distributed systems. CKML incorporates the CKP ideas of concept lattice and formal context, along with the IF ideas of classification (= formal context), infomorphism, theory, interpretation and local logic. Ontology Markup Language (OML), a subset of CKML that is a self-sufficient markup language in its own right, follows the principles and ideas of Conceptual Graphs (CG). OML is used for structuring the specifications and

axiomatics of metadata into ontologies. OML incorporates the CG ideas of concept, conceptual relation, conceptual graph, conceptual context, participants and ontology.

OML owes much to pioneering efforts of the SHOE initiative[57]. The earlier versions of OML were basically a translation to XML of the SHOE formalism, with suitable changes and improvements. Common elements could be described by paraphrasing the SHOE documentation. Later versions are RDF/Schemas compatible and incorporate a different version of the elements and expressiveness of conceptual graphs.

The Information Flow Framework (IFF)[73], a new markup language related to OML/CKML is currently being designed. It is founded upon the principles and techniques of Information Flow and Formal Concept Analysis, which incorporates aspects (classification, theories, logics, terminologies) from OML/CKML, OIL and RDF/S.

Ideas in OML/CKML and IFF are also at the base of the development of the SUO (Standard Upper Ontology) initiative[63].

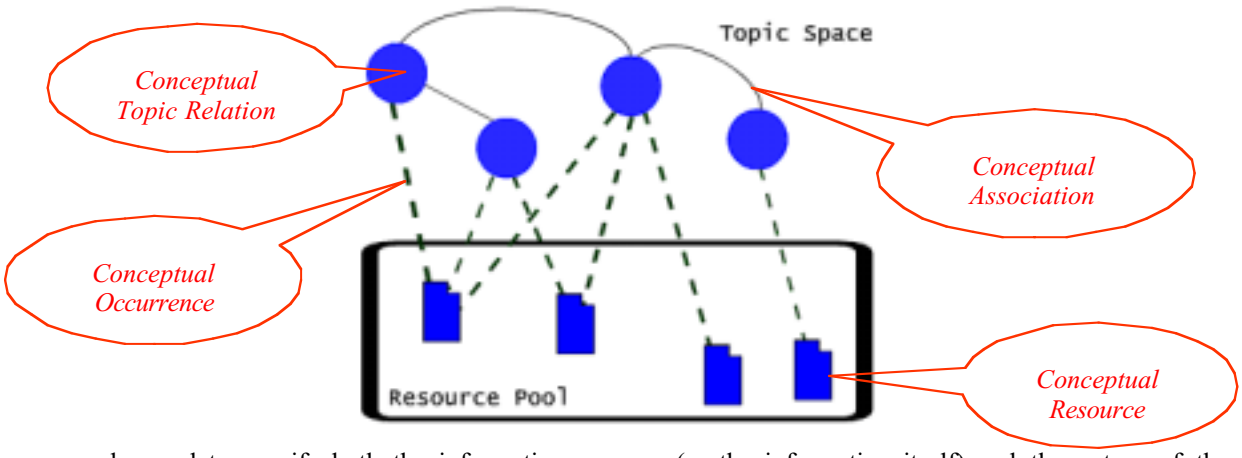
3.5.4 Topic Maps

WHAT to express:	Class/relation
HOW to express:	Graph Based
HOW to interpret the expression:	N/A (No formal semantics)

Topic Maps are a way of modelling a set of subjects, the relationships between them and additional information about them, with the same logics of a back-of-book index. The topic map paradigm is described in the ISO standard ISO 13250:2002[120]. The first edition of the ISO specification (2000), described an an interchange syntax (HyTM) based on SGML. Since then, a separate consortium, TopicMaps.Org[114], have produced a version of the topic map paradigm for use in a web environment. The XML Topic Maps (XTM) 1.0 specification[123] developed by TopicMaps.Org defines an interchange syntax based on XML and XLink. XTM has now been adopted by ISO and the ISO 13250 standard in its second edition now contains both syntaxes.

In essence a topic map can be thought of as consisting of just three things:

- *Topics* which describe subjects in the real world. A subject may be something that is addressable by a machine (e.g. a web-page with a URL), something which is not addressable (such as a person) or even an abstract concept (such as 'Music').
- *Associations* which describe the relationships between topics. The association construct may include additional typing information which specifies the nature of the relationship between the topics and also what role each topic plays in the relationship. So an association can be used to represent such common concepts as a hierarchy, or a group but also more complex relationships such as a meeting (with topics representing people and playing roles such as 'chairman', 'scribe' and 'attendee').
- *Occurrences* which connect a topic to some information resource related to the topic. The occurrence can



be used to specify both the information resource (or the information itself) and the nature of the relationship between the information resource and the topic. For example, a topic could represent a

person and that topic could have an occurrence which is a URL which points to the 'homepage' of the person.

Figure 1:Resource management in the topic maps model.

Furthermore, other important concepts are:

- *Topic classes, Occurrences classes, Association classes:* allow to group different kinds of topics, occurrences and associations, respectively. Topics, occurrences and associations are instances of such classes. It is important to note that these classes are topics as well. It is possible to define a taxonomic hierachy of these classes.
- *Scope:* a scope indicates the context within which a characteristic of a topic (a name, an occurrence or a role played in an association) may be considered to be true. One common use of scope is to provide localised names for topics. For example "company" in English could be "soci t " in French or "Firma" in German. They all describe the same concept and so should be represented by the same topic. Rather than creating separate topics one should instead create a single topic to represent the concept of a company and add to it three separate base names, using scope to indicate that a particular base name should be used for a specific language environment.

In addition to defining these basic structures, the topic map standards also define the way in which two or more topic maps may be combined or merged. Topic map merging is a fundamental and highly useful characteristic of topic maps as it allows a modular approach to the creation of topic maps and it provides a means for different users to share and combine their topic maps in a controlled manner.

Topic Maps are very flexible but difficult to evolve, maintain and keep consistent. Furthermore they have a non-well defined semantics. they represent an informal (or, at least, semi-formal) approach to knowledge representation. Some work is currently being done on a formal data model for topic maps (e.g. [88][4]). Despite this lack of formal semantics and the relatively new status of related standards (the ISO specification was published in January 2000, with the XTM specification being completed in February 2001), Topic Maps are used for several Knowledge Management applications. Commercial and non-commercial implementations of so-called 'topic map engines' are already being offered. These 'engines' have certain features in common all provide the means to persistently store data structured as topic maps and to retrieve that data programmatically via an API. Other commercial and non-commercial applications of those engines are also available, including applications to present topic map data in a user-friendly browser-interface and applications to manually or automatically create topic maps from data sources.

3.6 Logic-based languages

In this section, we introduce the class of languages that are based on logic. Such languages express a domain-ontology, i.e. a conceptualization of the world, in terms of the classes of objects that are of interest in the domain, as well as the relevant relationships holding among such classes.

These kinds of languages have a formal well-defined semantics. Intuitively, this means that, given exact meanings for the symbols of the language, i.e. exact referent in the set of all objects presumed or hypothesized to exist in the world, called *universe of discourse* (also called *domain of interpretation*), the semantics of the language tells the exact meaning of arbitrary sentences. Nevertheless, in keeping with traditional logical semantics, it is not possible to know the exact referent in the universe of discourse for every constant in the language, since everybody has his own, incomplete, perception of the objects of the world. Thus, the semantics of the language cannot pick out exact meanings for all expressions in the language, but it does place constraints on the meanings of complex expressions. Therefore, objects in the universe of discourse do not in themselves have any meaning, nor does the choice of any particular set of objects that make up the universe of discourse - what is important is the relationships between objects and sets of objects. Such relationships have to hold in all possible universe of discourse.

More formally, the semantics of a logic-based language is given via *interpretations*. An interpretation is a couple I=(UI,•I), where UI is the domain of interpretation and •I is a function that gives to all objects, combinations of objects and relationships of the language, a meaning in the domain UI. Given an ontology expressed in a logic-based language, a *model* is an interpretation that gives the ontology a meaning that is consistent with respect to the domain UI. An ontology in such languages describes objects and relationships that have to exist and hold respectively in all its possible models.

We will first present languages based on first-order predicate logic. Then we will discuss ontology languages based on Description Logics. Finally, process/action specification-languages will be introduced.

3.6.1 Based on first-order predicate logic

In what follows, we provide a brief presentation of the most common ontology languages that are based on first order predicate logic. An ontology in such languages is described in terms of constants, predicates, functions.

We will present two languages, KIF and CycL, that both provide an important extension of first-order logic, by allowing the reification of formulas as terms used in other formulas. Therefore, KIF and CycL allow meta-level statements. As a matter of fact, such languages are not intended to provide logical reasoning, since this would be necessarily undecidable.

3.6.1.1 KIF

WHAT to express:	Class/relation
HOW to express:	Logic-based
HOW to interpret the expression:	Several models

Knowledge Interchange Format (KIF)[44] is a formal language for the interchange of knowledge among disparate computer programs. Different programs can interact with their users in whatever forms are most appropriate to their application. A program itself can use for the computation whatever internal form it prefers for the data it manages. The idea is that when a program needs to communicate with another program, it maps its internal data structures into KIF. Therefore, KIF is not intended neither as a primary language for interaction with human users, nor as an internal representation for knowledge within computer programs. On the contrary, its main purpose is to facilitate the independent development of knowledge-manipulation programs.

Given the following basic features, KIF can be used as a representation language, though.

1. The language has a declarative semantics, which means that it is possible to understand the meaning of the expressions of the language without appeal to an interpreter.
2. The language is logically comprehensive, in that it provides for the expression of arbitrary sentences in the predicate calculus.
3. The language provides means for the representation of knowledge about the representation of knowledge.
4. The language provides for the definition of axioms for constants.
5. The language provides for the specification of non-monotonic reasoning rules.

More in details, a knowledge base in KIF consists of a set of *forms*, where a form is either a *sentence*, a *rule*, or a *definition*. Let us give an insight on these three types of forms. A **sentence**, which may be quantified universally or existentially, can be either a logical constant, an equation, an inequality, a relation constant associated to an arbitrary number of arguments or, finally, a combination of sentences by means of classic logical operators (negation, conjunction, disjunction, implication and equivalence). Such a sentence may contain (free) variables. A **definition,** for most purposes, can be viewed as shorthand for the sentences in the content of the definition. It allows to state sentences that are true by definition . A **rule**, that is called forward or reverse depending on the particular operator involved, can be monotonic or non-monotonic. Monotonic rules are similar to inference rules, familiar from elementary logic. Non-monotonic rules provide the user to express his non-monotonic reasoning policy within the language, in order to draw conclusions based, for example on the absence of knowledge.

Note that constants of the language can be either objects constants, function constants, relation constants or logical constants, that express conditions about the world and can be either true or false. There is no way of determining the category of a (non-basic) constant from its syntax. The distinction is entirely semantic, which reifies second-order features in KIF, by letting express statements about statements.

3.6.1.2 CycL

WHAT to express:	Class/relation

HOW to express:	Logic Based
HOW to interpret the expression:	Several models

Begun as a research project in 1984, Cyc [79] purpose was to specify the largest common-sense ontology that should provide Artificial Intelligence to computers. Such a knowledge base would contain general facts, heuristics and a wide sample of specific facts and heuristics for analogising as well. It would have to span human consensus reality knowledge: the facts and concepts that people know and those that people assume that others know. Moreover this would include beliefs, knowledge of respective limited awareness of what people know, various ways of representing things, knowledge of which approximations (micro-theories) are reasonable in various context, and so on. Far from having attained this goal, Cyc[1] is now a working technology with applications to real-world business problems. Cyc vast knowledge base enables it to perform well at tasks that are beyond the capabilities of other software technologies. At the present time, the Cyc KB contains nearly two hundred thousand terms and several dozen hand-entered assertions about/involving each term.

CycL is the language in which Cyc knowledge base is encoded. It is a formal language whose syntax derives from first-order predicate calculus and from Lisp. In order to express common sense knowledge, however, it extends first order logic, with extensions to handle equality, default reasoning, skolemization, and some second-order features. For example, quantification over predicates is allowed in some circumstances, and complete assertions can appear as intensional components of other assertions. Moreover, CycL extends first-order logic by typing, i.e. functions and predicates are typed.

The vocabulary of CycL consists of terms. These are combined into *assertions*, which constitute the knowledge base. The set of terms can be divided into *constants*, *non-atomic terms*, *variables*, and a few other types of objects (numbers, strings,). More precisely, **constants** are the "vocabulary words" CycL. Constants may denote either individuals, collections of other concepts (i.e. sets of the individuals identified by means of unary predicates), arbitrary predicates that allow to express relationships among constants, or functions, which can take constants or other things as arguments, and generate new concepts. A **non-atomic term** is a way of specifying a term as a function of some other term(s). Every non-atomic term is composed of a function and one or more arguments to that function. **Variables** stand for constants whose identities are not specified. Finally, an **assertion** is the fundamental unit of knowledge in the Cyc Knowledge base. Every assertion consists of:

- a *CycL formula* which states the content of the assertion; a **CycL formulas** has the structure of a Lisp list: it consists of a list of objects, the first of may be a predicate, a logical connective, or a quantifier; the other objects may be atomic constants, non-atomic terms, variables, numbers, strings, or other formulas; to be considered a *well-formed* formula all the variables it includes need to be bound by a quantifier before being used;
- a *truth value* which indicates its degree of truth; there are five possible truth values, of which the most common are the first two:
    - **monotonically true**: true always and under all conditions;
    - **default true**: assumed true, but subject to exceptions;
    - **unknown**: not known to be true, and not known to be false;
    - **default false**: assumed false, but subject to exceptions;
    - **monotonically false**: false always and under all conditions;
- a *microtheory* of which the assertion is part; a **microtheory,** also called *contexts*[101], denotes a set of assertions which are grouped together because they share a set of assumptions;
- a *justification*; a **justification** refers to a reason why the assertion is present in the KB with its truth value; it may be a statement that tells that the formula was explicitly "asserted", or it may be the group of assertions through which the assertion was inferred.

[11] http://www.cyc.com

3.6.2 Based on description logics

Description Logics (DLs) are a family of logic-based knowledge representation formalisms designed to represent and reason about the knowledge of an application domain in a structured and well-understood way. The basic notions in DLs are concepts and roles, which denote sets of objects and binary relations, respectively. Complex concept expressions and role expressions are formed by starting from a set of atomic concepts and atomic roles, i.e., concepts and roles denoted simply by a name, and applying concept and role constructs. Thus, a specific DL is mainly characterized by the constructors it provides to form such complex concepts and roles. DLs differ from their predecessors, such as semantic networks (cf. Section 3.5.1) and frames (cf. Section 3.2), in that they are equipped with a formal, logic-based semantics. Such well-defined semantics and implemented powerful reasoning tools make DLs ideal candidates for ontology languages [6].

In this section, we will present three classes of languages based on description logics. For each of them, we will discuss more deeply some representative languages. Firstly, we will consider languages based on pure description logics. Secondly, we will introduce a class of hybrid languages, combining description logics and Horn rules. Thirdly, we will present semantic web ontology languages, whose semantics can be defined via a translation into an expressive DL. An important feature of all the languages that will be discussed is that they come with reasoning procedures that are sound and complete with respect to a specified semantics, i.e. all the sentences that can be inferred from the ontology are true (soundness), and given an ontology, if there is a sentence that is true, then such a sentence can be inferred (completeness).

Observe that, in principle, as long as the trade-off between expressivity of the language and complexity of its reasoning capabilities is investigated and solved with respect to the application domain, any DL language may be used to represent an ontology. However, it is far beyond the scope of this survey to present all the DL languages. For a comprehensive discussion on DLs, see [5][119].

3.6.2.1 Pure Description Logic languages.

An ontology can be expressed in a DL by introducing two components, traditionally called TBox and ABox (originally from Terminological Box and Assertional Box respectively)[2]. The TBox stores a set of universally quantified assertions, stating general properties of concepts and roles. Indeed concepts (or roles) are defined intensionally, i.e. in terms of descriptions that specify the properties that objects must satisfy to belong to the concept (or pairs of objects must satisfy to belong to the role). The ABox comprises assertions on individual objects, called instance assertions, that consist basically of memberships assertions between objects and concepts (e.g. a is an instance of C), and between pairs of objects and roles (e.g. a is related to b by the role R). Note that what differentiates one DL from another is the kinds of constructs that allow the construction of composite descriptions, including restrictions on roles connecting objects.

Several reasoning tasks can be carried out on a knowledge base of the above type, in order to deduce implicit knowledge from the explicitly represented knowledge. The simplest forms of reasoning involve computing: i) *subsumption* relation between two concept expressions, i.e. verifying whether one expression always denotes a subset of the objects denoted by another expression, ii) *instance-of* relation, i.e. verifying whether an object o is an element denoted by an expression, and iii) *consistency algorithm*, i.e. verifying whether a knowledge base is non-contradictory.

In order to ensure a reasonable and predictable behaviour of a DL system, these inference problems should at least be decidable for the DL employed by the system, and preferably of low complexity. Consequently, the expressive power of the DL in question must be restricted in an appropriate way. If the imposed restrictions are too severe, however, then the important notions of the application domain can no longer be expressed. Investigating this trade-off between expressivity of DLs and the complexity of their deduction capabilities has been one of the most important issues in DL research. As an example of basic DL, we introduce the language ALC in what follows.

3.6.2.1.1 The language ALC

WHAT to express:	Class/relation

[2] Since TBox and ABox constitute two different components of the same knowledge base, sometimes [41] this kind of systems have been also called hybrid. However, we prefer here to distinguish between terminological/assertional hybrid systems such as this one, in which one of the components may be empty, that we call pure DL systems, and more general hybrid systems that will be

HOW to express:	Logic Based
HOW to interpret the expression:	Several models

ALC[104] let form concept descriptions starting from atomic concepts, the universal concept (_) and the bottom concept (⊥), and using basic set operators, namely set complement, intersection and union. It permits a restricted form of quantification realized through so-called quantified role restrictions, that are composed by a quantifier (existential or universal), a role and a concept expression. Quantified role restriction allow one two represent the relationships existing between objects in two concepts. For example one can characterize the set of objects all of whose children are blond as **∀child.Blond** as well as the set of objects that have at least one blond child **∃child.Blond**. The former construct is called universal role restriction, or also value restriction, while the latter is called qualified existential role restriction.

From the point of view of semantics, concepts are interpreted as subsets of a domain (the universe of discourse) and roles as binary relations over that domain. An interpretation **I**=(**UI**,•**I**) over a set **A** of atomic concepts and a set **P** of atomic roles consists of a non-empty set **UI** (also called domain of **I**) and a function •**I** (called interpretation function of **I**) that maps every concept **a** ∈ **A** to a subset **AI** of **UI** (the set of instances of **A**) and every atomic role **p** ∈ **P** to a subset **PI** of **UI** × **UI** (the set of instances of **P**). The bottom concept is always mapped to the empty set, while the universal concept is mapped to **UI**. The interpretation function can then be extended to concepts descriptions, inductively, as it follows: (i) the negation of a concept is mapped to the complement of the interpretation of the concept, (ii) the intersection of two concepts is mapped to the intersection of the interpretations of the two concepts, (iii) the union of two concepts is mapped to the union of the interpretations of the two concepts, (iv) the universal role restriction **∀R.C** is mapped to the subset of elements a of **UI** such that if **a** is related to **b** by means of the interpretation of **R** (**RI**), then **b** belongs to the interpretation of **C**, (v) the qualified existential role restriction **∃R.C** is mapped to the subset of elements a of **UI** that are related to an element b of the interpretation of **C** by means of **RI**.

3.6.2.2 Hybrid languages: AL-Log, CARIN.

Hybrid systems are a special class of knowledge representation systems which are constituted by two or more subsystems dealing with distinct portions of a knowledge base and specific reasoning procedures. Their aim is to combine the knowledge and the reasoning of different formalisms in order to improve representational adequacy and deductive power. Indeed, in exchange for a quite expressive description of the domain of interest, DLs pay the price of a weak query language. Indeed, a knowledge base is able to answer only queries that return subsets of existing objects, rather than creating new objects; furthermore, the selection conditions are rather limited. Given such expressive limitations of DL concepts alone as queries, it is reasonable to consider extending the expressive power of standard queries, i.e. Horn Rules, with the structuring power of DLs. In what follows, we will present two languages that have pursued such an issue, each of them is used to specify knowledge in two different hybrid systems.

3.6.2.2.1 AL-Log

WHAT to express:	Class/relation
HOW to express:	Logic Based
HOW to interpret the expression:	Several models

AL-Log[30] is a system constituted by two subsystems, called structural and relational. The former is based on the DL language ALC and allows one to express knowledge about concepts, roles and individuals. It can be itself regarded as a terminological/assertional system. The latter provides a suitable extensions of Datalog in order to express a form of relational knowledge. The interaction between the two subsystems is realized by allowing the specification of constraints in the Datalog clauses, where constraints are expressed using *ALC*.. The constraints on the variables require them to range over the set of instances of a specified concept, while the constraints on individual objects require them to belong to the extension of a concept. Therefore DLs is used as a typing language on the variables appearing in the rules, and only unary predicates from the DLs are allowed in the rules.

3.6.2.2.2 CARIN

WHAT to express:	Class/relation
HOW to express:	Logic Based
HOW to interpret the expression:	Several models

CARIN[81] is a family of representation languages parameterised by the specific DL used, provided it is any subset of the expressive DL ALCNR [16], which extends ALC, by *unqualified number restrictions*, i.e. the possibility to constrain the number of arbitrary objects that are in relationships with a given object, and by intersection of roles. A CARIN-L knowledge base contains three parts: i) a *terminology T*, i.e. a terminological component expressed in L, ii) a *rule* base *R*, i.e. a set of Horn rules, and iii) a set *A* of *ground facts* for the concepts, roles and predicates appearing in *T* and *R*. CARIN treats concepts and roles as ordinary unary and binary predicates that can also appear in query atoms. This is significant because it allows for the first time conjunctive queries to be expressed over DL ABoxes.

3.6.2.3 Semantic web ontology languages.

Several ontology languages have been designed for use in the Web. Among them, the most important are OIL [37], DAML-ONT[58] and DAML+OIL[26][85]. More recently, a new language, OWL [29], is being developed by the World Wide Web Consortium (W3C) Web Ontology Working Group, which had to maintain as much compatibility as possible with pre-existing languages and is intended to be proposed as the standard Semantic Web ontology language. The idea of the semantic Web is to annotate web pages with machine-interpretable description of their content. In such a context, ontologies are expected to help automated processes to access information, providing structured vocabularies that explicate the relationships between different terms.

Except for DAML-ONT, that was basically an extension of RDF (cf. Section 3.7.2.1) with language constructors from object-oriented and frame-based knowledge representation languages (cf. Section 3.2), and, as such, suffered from an inadequate semantic specification, the above mentioned ontology languages for the Web are all based on DLs. We chose here to focus on the description of OWL, instead of presenting all its predecessors. Refer to [60] or a comprehensive survey on the making of OWL from its predecessor.

3.6.2.3.1 OWL

WHAT to express:	Class/relation
HOW to express:	Logic Based
HOW to interpret the expression:	Several models

Intuitively, OWL can represent information about categories of objects and how objects are interrelated. It can also represent information about objects themselves. More precisely, on the one side, OWL let describe **classes** by specifying relevant properties of objects belonging to them. This can be achieved by means of a *partial* (necessary) or a *complete* (necessary and sufficient) definition of a class as a logical combination of other classes, or as an enumeration of specified objects. OWL let also describe **properties** and provide domains and ranges for them. Domains can be OWL classes, whereas ranges can be either OWL classes or externally-defined datatypes (e.g. string or integer). Moreover, OWL let provide restrictions on how properties behave that are local to a class. Thus, it is possible to define classes where a particular property is restricted so that (i) all the values for the property in instances of the class must belong to a certain class or datatype (universal restriction), (ii) at least one value must come from a certain class or datatype (existential restriction), and (iii) there must be at least or at most a certain number of distinct values (cardinality restriction). On the other side, OWL can also be used to restrict the models to meaningful ones by organizing classes in a subclass hierarchy, as well as properties in a subproperty hierarchy. Other features are the possibility to declare properties as transitive, symmetric, functional or inverse of other properties, and couple of classes or properties as disjoint or equivalent. Finally, OWL represents information about individuals, providing axioms that state which objects belong to which classes, what the property values are of specific objects and whether two objects are the same or are distinguished.

OWL is quite a sophisticated language. Several different influences were mandated on its design. The most important are DLs, the frames paradigm and the Semantic Web vision of a stack of languages including XML and RDF. On the one hand, OWL semantics is formalised by means of a DL style model theory. In particular, OWL is based on the SH family of Description Logics [61], which is equivalent to the ALC DL (cf

Section 3.6.2.1.1) extended with transitive roles and role hierarchies. Such family of languages represents a suitable balance between expressivity requirements and computational ones. Moreover, practical decision procedures for reasoning on them are available, as well as implemented systems such as FaCT[59] and RACER[55]. On the other hand, OWL formal specification is given by an abstract syntax, that has been heavily influenced by frames and constitutes the surface structure of the language. Class axioms consist of the name of the class being described, a modality indicating whether the definition of the class is partial or complete, and a sequence of property restrictions and names of more general classes, whereas property axioms specify the name of the property and its various features. Such a frame-like syntax makes OWL easier to understand and to use. Moreover, axioms can be directly translated into DL axioms and they can be easily expressed by means of a set of RDF triples (cf. Section 3.7.2.1). This property is an essential one, since OWL was also required to have RDF/XML exchange syntax, because of its connections with the Semantic Web.

Given the huge number of requirements for OWL and the difficulty of satisfying all of them in combination, three different versions of OWL have been designed:

- OWL DL, that is characterized by an abstract frame-like syntax and a decidable inference; it is based on SHOIN(**D**) DL, which extends SH family of languages with inverse roles, nominals (which are, in their simplest form, special concept names that are to be interpreted as singleton sets), unqualified number restrictions and support for simple datatypes; SHOIN(**D**) is very expressive but also difficult to reason with, since inference problems have NEXPTIME complexity;
- OWL Lite, that constitutes a subset of OWL DL that is similar to SHIF(**D**) and, as such, it does not allow to use nominals and allows only for unqualified number restrictions in the form **†1 R**; inference problems have EXPTIME complexity and all OWL DL descriptions can be captured in OWL Lite, except those containing either individuals names or cardinalities greater than 1;
- OWL Full, that, unlike the OWL DL and OWL Lite, allows classes to be used as individuals and the language constructors to be applied to the language itself; it contains OWL DL but goes beyond it, making reasoning undecidable; moreover, the abstract syntax becomes inadequate for OWL Full, which needs all the official OWL RDF/XML exchange syntax expressivity.

3.6.2.4 Process/Action specification languages.

A fundamental problem in Knowledge Representation is the design of a logical language to express theories about actions and change. One of the most prominent proposals for such a language is John McCarthy's situation calculus[84], a formalism which views situations as branching towards the future. Several specification and programming languages have also been proposed, based on the situation calculus. GOLOG[80], maintains an explicit representation of the dynamic world being modeled, on the basis of user supplied axioms about the preconditions and effects of actions and the initial state of the world. This allows programs to reason about the state of the world and consider the effects of various possible courses of action before committing to a particular behavior. The next section describes the Process Specification Language PSL.

3.6.2.4.1 PSL

WHAT to express:	Process/Action
HOW to express:	Logic based
HOW to interpret the expression:	Several models

Process Specification Language (PSL) [50][51] is a first order logic ontology explicitely designed for allowing (correct) interoperability among heterogenous software applications, that exchange information as first-order sentences. It has undergone years of development in the business process modeling arena, by defining concepts specifically regarding manufacturing processes and business process. Recently, PSL has become an International Standard (ISO 18629).

PSL can model both black box processes, i.e., activities, and white box processes, i.e., complex activities, and allows formulae that explicitly quantify over and specify properties about complex activities. The latter aspect, not shared by several other formalisms, makes it possible to express in an explicit manner broad variety of properties and constraints on composite activities.

PSL is constituted by (finite) set of first order logic theories defining concepts (i.e., relations and functions) needed to formally describe process and its properties (e.g., temporal constraints, activity occurrences, etc.): they are defined starting from primitive ones, whose meaning is assumed in the ontology. The definitions

are specified as set of axioms expressed using the Knowledge Interchange Format, KIF (cf. Section 3.6.1.1). PSL is extensible, in the sense that other theories can be created in order to define other concepts: of course, the newly added theories must be consistent with the core theories. Among the latter, Tpslcore defines basic ontological concepts, such as activities, activity occurrences, timepoints, and objects that participate in activities. Additional core theories (see Figure 2) capture the basic intuitions for the composition of activities, and the relationship between the occurrence of complex activity and occurrences of its subactivities. Among those, it is worthwhile to mention the following ones. Tocctree characterizes an occurrence tree, i.e., a partially ordered set of activity occurrences3. Tdisc_state defines the notion of fluents (state), which are changed only by the occurrence of activities. In addition, activities have preconditions (fluents that must hold before an occurrence) and effects (fluents that always hold after an occurrence). Tcomplex characterizes the relationship between the occurrence of complex activity and occurrences of its subactivities: an occurrence of complex activities corresponds to an occurrence tree, i.e., a partially ordered set of occurrences of subactivities. Finally, Tactocc captures the concepts related to occurrences of complex activities, and, in particular, activities trees and their relations which occurrences trees (activities trees are part of occurrence trees).

Figure 2: Primitive Lexicon of PSL

Figure 2 illustrates (part of) the lexicon of PSL and its intuitive meaning. Most of the terms are

$T_{pslcore}$	$activity(a)$	a is an activity
	$activity\_occurrence(o)$	o is an activity occurrence
	$timepoint(t)$	t is a timepoint
	$object(x)$	x is an object
	$occurrence\_of(o, a)$	o is an occurrence of a
$T_{occtree}$	$successor(a, s)$	the element of an occurrence tree that is the next occurrence of a after the activity occurrence s
	$legal(s)$	s is an element of a legal occurrence tree
	$initial(s)$	s is the root of an occurrence tree
	$earlier(s_1, s_2)$	s_1 precedes s_2 in an occurrence tree
	$poss(a, s)$	there exists a legal occurrence of a that is a successor of s
$T_{discstate}$	$holds(f, s)$	fluent f is true immediately after the activity occurrence s
	$prior(f, s)$	fluent f is true immediately before the activity occurrence s
$T_{complex}$	$min_precedes(s_1, s_2, a)$	the atomic subactivity occurrence s_1 precedes the atomic subactivity occurrence s_2 in an activity tree for a
	$root(s, a)$	the atomic subactivity occurrence s is the root of an activity tree for a
T_{actocc}	$subactivity\_occurrence(o_1, o_2)$	o_1 is a subactivity occurrence of o_2
	$root\_occ(o)$	the initial atomic subactivity occurrence of o
	$leaf\_occ(s, o)$	s is the final atomic subactivity occurrence of o

primitive fluents, with the following exceptions: successor (a, s) is primitive function, poss(a, s), root_occ(o) and leaf_occ(s, o) are defined in their respective theory (as KIF axiom). For example, the axiom capturing poss(a, s) is:

(forall (?a ?s) (iff (poss ?a ?s)

[3] Occurrence trees are isomorphic to the situation trees that are models of the axioms for Situation Calculus: from this PSL can be considered as a dialect of Situation Calc

(legal (successor ?a ?s))))

It states that it is possible to execute an activity a after activity occurrence s if and only if the successor occurrence of the activity a is legal. Note that legal(Æ) and successor(Æ) are primitive concepts.

As far as its semantics, PSL has a model-theoretic semantics, based on trees that represent possible sequences of (atomic) activities that might occur in an execution of an application. The meaning of terms in the ontology is characterized by models for first-order logic. Indeed, the model theory of PSL provides rigorous abstract mathematical characterization of the semantics of the terminology of PSL. This characterization defines the meanings of terms with respect to some set of intended interpretations represented as class of mathematical structures. Furthermore, for each theory in PSL, there are theorems that prove that every intended interpretation is model of the theory, and every model of the theory is isomorphic to some intended interpretation. Therefore, PSL has first-order axiomatization of the class of models, and classes in the ontology arise from classification of the models with respect to invariants (properties of the models preserved by isomorphism). Note that PSL semantics is based on the notion of tree, therefore, it is second order. However, because of the assumption that processes exchange only first order sentences, PSL s syntax is first order. This implies that PSL cannot express rechability properties (i.e., after activity occurrence a, eventually activity b occurs), since second order formula is needed.

The following example, taken from [50], illustrates some sentences in PSL.

Example:

$\forall$(x, o) occurrence_of(o, drop(x)) $\supset$ prior(fragile(x),o) $\wedge$ holds(broken(x),o)

Intuitively, this formula states that if the object x is fragile, (i.e., fragile(x) is true immediately before activity drop(x) occurs) then x will break when dropped (i.e., broken(x) is true immediately after activity drop(x) occurs). [4]

$\forall$(x, o) occurence_of(o, make(x)) $\supset$ ($\exists$ o1, o2, o3) occurrence_of (o1 ,cut(x))

$\wedge$ occurrence_of (o2, punch(x)) $\wedge$ occurence_of (o3, paint(x))

$\wedge$ subactivity_occurrence(o1,o) $\wedge$ subactivity_occurrence(o2,o)

$\wedge$ subactivity_occurrence(o3,o)

$\wedge$ min_precedes(leaf_occ(o1), root_occ(o3),make(x))

$\wedge$ min_precedes(leaf_occ(o2), root_occ(o3),make(x))

Intuitively, this sentence states that in order to make the frame (activity make(x)), the cutting (activity cut(x)) and the punching (activity punch(x)) should be done in parallel, before doing the painting (activity paint(x)). Specifically, if o is the occurrence of activity make(x), then there exists o1,o2 and o3 which are occurrences of cut(Æ), punch(Æ)and paint(Æ), respectively, and which are subactivities of o. There is a partial ordering constraint between o1,o2 and o3: it is imposed that the last atomic subactivities of o1 and o2 take place before the firts atomic subactivity of o3; the parallelism constraint is captured by imposing no ordering constraint on the occurrences of subactivities of o1 and o2. Finally, note that here we are implicitely assuming that the activities are complex, i.e., they are constituted by certain number of subactivities. If this is not the case, i.e., o1 and o2 are primitive5, the fluents min_precedes have o1 and o2 as first component (instead of their roots), and the conjunction primitive(o1) $\wedge$ primitive(o2)6 is added to the above sentence.

[4] The frame problem is implicitly solved by the semantics of *holds*.

[5] Note that PSL makes a distinction between an atomic and a primitive activity. An activity is called primitive if it has no subactivities, it is atomic if either primitive or it is the concurrent aggregation of primitive activities.

[6] The term *primitive(.)* is defined in the Subactivity theory that is not repo

3.7 XML-related formalisms

XML[7] is a tag-based language for describing tree structures with a linear syntax. It has been developed by the W3C XML Working Group and has become the standard de-facto language for exchange of information in the Web. It is *self-describing*, since by introducing tags and attribute names on documents, it defines their structure while providing semantic information about their content.

Given the popularity of XML in exchange of information, XML—related languages have been considered as suitable for ontology representation (and exchange). In this section we examine how such an issue has been pursued. As for logic-based languages, a domain ontology expressed in XML-related languages describes the domain of interest in terms of classes of objects and relationships between them. We will first present two languages for the *validation* of XML documents: DTD and XML-Schema languages. Then we will introduce RDF and RDFS, whose aim is to provide a foundation for processing metadata about Web documents.

3.7.1 Languages for the validation of XML documents.

Two main languages have been developed to define the structure and constraints of *valid* XML documents, i.e. to basically define the legal nesting of elements and the collection of attributes for them. Such languages are called Document Type Definition (DTD) and XML-Schema. They can be used to some extent to specify ontologies. From the point of view of semantics, an ontology expressed by means of these languages is interpreted with respect to one logical model, since an ontology described either in a DTD, or in an XML-Schema, represents basically a document instance, i.e. a valid XML document.

3.7.1.1 DTD

WHAT to express:	Everything
HOW to express:	XML syntax
HOW to interpret the expression:	Single model

Since a DTD defines a valid tree structure, it can be used to express an ontological hierarchy. Element attributes can represent objects properties, while by means of element nesting it is possible to express subclass as well more sophisticated relationships between classes of objects. Nevertheless, several problems arises trying to express an ontology with a DTD. Among them, the most notable concern with (i) the need to reify attribute names with class names to distinguish with different appearance of the same attribute name in various concepts, (ii) the need to make subelements ordering irrelevant, (iii) poor means for defining semantics of elementary elements (the only primitive datatype in a DTD is PCDATA, i.e. arbitrary strings), and (iv) the lack of inheritance notion in DTDs. In a nutshell, DTDs are rather weak in regard to what can be expressed with them. Moreover, DTDs do not provide any means More details on ontology description by means of DTDs can be found in [32][100].

3.7.1.2 XML-Schema

WHAT to express:	Everything
HOW to express:	XML syntax
HOW to interpret the expression:	N/A (No formal semantics) ? Single model?

XML-Schema[122] provides a means for defining the structure, content and semantics of XML documents. An easily readable description of the XML Schema facilities can be found in [35]. For a formalization of the essence of XML-Schema see [106].

[7] http://www.w3.org/XML/

As for the DTD (cf. Section 3.7.1.1), the purpose of XML Schema is therefore to declare a set of constraints that an XML document has to satisfy in order to be validated. With respect to DTD, however, XML-Schema provides a considerable improvement, as the possibility to define much more elaborated constraints on how different part of an XML document fit together, more sophisticated nesting rules, data-typing. Moreover, XML-Schema expresses shared vocabularies and allow machines to carry out rules made by people. Among a large number of other rather complicated features (XML-Schema specification consists of more than 300 pages in printed form!), we finally mention the possibility, offered by the namespace mechanism, to combine XML documents with heterogeneous vocabulary.

As XML-Schema provides vocabulary and structure for describing information sources that are aimed at exchange, it can be used as an ontology language. However, in XML Schema all inheritance must be defined explicitly, so reasoning about hierarchical relationships is not an issue, and XML-Schema can be only considered as an ontology representation language. For a comprehensive study of the relations existing between ontologies and XML-Schema see [76].

3.7.2 Languages for representing information in the Web.

As the Web is huge and is growing at a healthy pace, there is the need to describe Web resources, by means of metadata that applications may exchange and process. The Resource Description Framework (RDF) and its *ontology-style* specialization, the Resource Description Framework Schema (RDFS), have both the purpose to provide a foundation for representing and processing metadata about Web documents. This would lead to a characterization of the information in the Web that would let reason on top of the largest body of information accessible to any individual in the history of the humanity.

3.7.2.1 RDF (RDFS)

WHAT to express:	Class/relation
HOW to express:	XML syntax
HOW to interpret the expression:	Several models

The Resource Description Framework (RDF) [77] is an XML application (i.e., its syntax is defined in XML) customized for adding semantics to a document without making any assumptions about the structure of the document. It is particularly intended for representing metadata about Web resources, such as the title, author, and modification date of a Web page, copyright and licensing information about a Web document, or the availability schedule for some shared resource. Thus, it enables the encoding, exchange and reuse of structured meta data, by providing a common framework for expressing this information so it can be exchanged between applications without loss of meaning. Search engines, intelligent agents, information broker, browsers and human user can make use of semantic information. However RDF can also be used to represent information about things that can be *identified* on the Web, even when they cannot be directly *retrieved* on the Web.

The RDF data model provides three object types: *resources*, *properties*, and *statements*:

- A *resource* may be either entire Web page, a part of it, a whole collections of pages, or an object that is not directly accessible via the Web;
- A *property* is a specific aspect, characteristics, attribute, or relation used to describe a resource.
- A *statement* is a triple consisting of two nodes and a connecting edge. These basic elements are all kinds of *RDF resources*, and can be variously described as <things> <properties> <values> [10], <object><attribute><value> [82], or <subject> <predicate> <object> [15]. According to the latter description, a *subject* is a resource that can be described by some property. For instance a property like *name* can belong to a dog, cat, book, plant, person, and so on. These can all serve the function of a subject. The *predicate* defines the type of property that is being attributed. Finally, the *object* is the value of the property associated with the subject.

The underlying structure of any expression in RDF is a therefore a collection of *node-arc-node* triples (i.e. statements). A set of such triples is called an *RDF graph*. . The assertion of an RDF graph amounts to asserting all the triples in it, so the meaning of an RDF graph is the conjunction of the statements corresponding to all the triples it contains. Getting in the heart of RDF statements, a node may be a *URIref* (standard URI with optional fragment identifier URI reference), a *blank node node* (placeholder for not-yet existing URI), or *literal* (value from a concrete domain; string, integer, etc.)., a literal, or blank (having no separate form of identification). In the RDF abstract syntax, a blank node is just a unique node that can be used in one or more RDF statements, but has no intrinsic name. Literals are used to identify values such as numbers and dates by means of a lexical representation. Anything represented by a literal could also be represented by a URI, but it is often more convenient or intuitive to use literals. A literal may be the object of an RDF statement, but not the subject or the predicate

Figure 3: Example of RDF graph

The example in Figure 3 from the RDF Primer[82] shows how simple RDF graphs can be combined to form full descriptions of resources. The graph also shows that literals are drawn by rectangles. Note also that literal values can only be used as RDF objects and never subjects or predicates.

RDF has a formal model-theoretic semantics[56] which provides a dependable basis for reasoning about the meaning of an RDF expression. Important aspects of such a formalization are:

- RDF supports rigorously defined notions of entailment which provide a basis for defining reliable rules of inference in RDF data.
- The specifications of RDF propose also an alternative way to specify a semantics, that is to give a translation from RDF into a formal logic with a model theory already attached. This 'axiomatic semantics' approach has been suggestedand used with various alternative versions of the target logical language [83][85]. Anyway, this alternative is given with the notice that in the event that any axiomatic semantics fails to conform to the model-theoretic semantics described for RDF, the model theory should be taken as normative.
- RDF semantics is defined associating interpretations to Vocabularies. A Vocabulary is the set of names (URIs and literals) that appear in an RDF Graph. Ground RDF Graphs, that is graphs that do not contain blank nodes, may be given a truth value with regard to an interpretation of their vocabulary. Blank nodes are treated as simply indicating the existence of a thing, without using, or saying anything about, the name of that thing. That is, all blank nodes are treated as having the same meaning as existentially quantified variables in the RDF graph in which they occur, and which have the scope of the entire graph. Unfortunately, some parts of the RDF and RDFS vocabularies are not assigned any formal meaning, as for the notable cases of the reification and container vocabularies.

- RDF is an assertional logic, in which each triple expresses a simple proposition. This imposes a fairly strict monotonic discipline on the language, so that it cannot express closed-world

assumptions, local default preferences, and several other commonly used non-monotonic constructs.

In order to use RDF as a means of representing knowledge it is necessary to enrich the language in ways that fixes the interpretation of parts of the language. As described thus far, RDF does not impose any interpretation on the kinds of resources involved in a statement beyond the roles of subject, predicate and object. It has no way of imposing some sort of agreed meaning on the roles, or the relationships between them. The RDF schema is a way of imposing a simple ontology on the RDF framework by introducing a system of simple types.

3.7.2.2 RDFS

WHAT to express:	Class/relation
HOW to express:	XML syntax
HOW to interpret the expression:	Single model

RDF Schema (RDFS) [14] enriches the basic RDF model, by providing a vocabulary for RDF, which is assumed to have a certain semantics. Predefined properties can be used to model *instance of* and *subclass of* relationships as well as domain restrictions and range restrictions of attributes. Indeed, the RDF schema provides modeling primitives that can be used to capture basic semantics in a domain neutral way. That is, RDFS specifies metadata that is applicable to the entities and their properties in all domains. The metadata then serves as a standard model by which RDF tools can operate on specific domain models, since the RDFS metamodel elements will have a fixed semantics in all domain models.

RDFS provides simple but powerful modeling primitives for structuring domain knowledge into classes and sub classes, properties and sub properties, and can impose restrictions on the domain and range of properties, and defines the semantics of containers.

The simple meta modeling elements can limit the expressiveness of RDF/S. Some of the main limiting deficiencies are identified in [75]:

- *Local scope of properties*: in RDFS it is possible to define a range on properties, but not so they apply to some classes only. For instance the property eats can have a range restriction of food that applies to all classes in the domain of the property, but it is not possible to restrict the range to plants for some classes and meat for others.
- *Disjointness* of classes can not be defined in RDF/S.
- *Boolean combinations of classes* is not possible. For example person cannot be defined as the union of the classes male and female.
- *Cardinality restrictions* cannot be expressed.
- *Special characteristics of properties* like transitivity cannot be expressed.

Because of these inherent limitations, RDF/S has been extended by more powerful ontology languages. The currently endorsed language is the Web Ontology Language (OWL) which has been discussed in Section 3.6.2.3.1.

3.8 Visual languages

In the last year several proposals have been presented, concerning methodologies and tools for ontology browsing and querying, focusing on visualization techniques. For instance in [90] the authors present ZIMS (Zoological Information Management System), a tool for generating graph based web representations of both data and metadada belonging to an ontology. The authors apply graph layout algorithm that attempts to minimize the distance of related nodes while keeping all the other nodes evenly distributed. The resulting graph drawings provide the user with insights that are hard to see in text.

Built on AT&T's Graphviz graph visualization software, IsAViz [98]˚is a stand alone application for browsing and authoring RDF documents. In addition to showing instance data, IsAViz shows connections between instances and˚ their originating classes. Though the graphs it generates are suitable for the intended task, because of their layout, they are not very useful while dealing with the overall structure of a set of instances.

The Prot g ontology editor, produced at Stanford University, is one of the more popular open source semantic web tools available today. It is easily extensible, and has two visualization components. OntoViz [95] is an ontology visualizer that shows classes and instances grouped with their properties; Groups are connected by edges denoting relationships between the objects. Jambalaya, another Prot g based visualization tool that shares with SEWASIE the goal of providing a view of the overall structure of ontologies, displays information similarly, with instances and classes being grouped with their respective properties. Jambalaya provides a zooming feature, allowing users to look at the ontology at different levels of detail. Both tools are designed to allow the user for˚ browsing an ontology but, because of they use a visual structure that shows all of the information associated with a specific object as nodes, the overall picture is full of large boxes, overlapped with edges, obscuring much of the associative structure.

The Spectacle system [40] is designed to present data instances to users in a way that allows the user for easy data navigation. Exploiting class memberships, instances are arranged in clusters; instances that belong to multiple classes are placed in overlapping clusters. This visualization provides a clear and intuitive representation of the relationships between instances and classes, as well as illustrates the connections between classes as they overlap. In many cases, though, grouping by classes is not the best choice for understanding the underlying data structure. As an example, when looking at data about people, projects, and papers produced by an organization, it is not useful to see people, projects, and papers grouped together. What is more interesting is to see clusters of people based on their interaction: small clusters that group people and papers around their respective projects say much more about how the organization is structured.

While the above proposals focuses on browsing activities, the one hosted on the Ontoweb website [96] are intended for expressing queries as well. In particular in [64] the authors describe an interactive "focus + context" visualization technique to support effective search. They have implemented a prototype that integrates WordNet, a general lexical ontology, and a large collection of professionally annotated images. As the user explores the collection, the prototype dynamically changes the visual emphasis, the detail of the images,˚ and the keywords, to reflect the relevant semantic relationships. This interaction allows the user to rapidly learn and use the domain knowledge required for posing effective queries.

An interesting survey is presented in [42], discussing the different user interface tactics that can be implemented with the help of ontologies in the context of information seeking and showing several experiences about ontology-based search interfaces based on thesauri [43].

The proposal presented in [78] describes a new approach for representing the result of a query against hierarchical data. An interactive matrix display is used for showing the hyperlinks of a site search or other search queries in different hierarchical category systems. The results of a site search are not only shown as a list, but also classified in an ontology-based category system. So the user has the possibility to explore and navigate within the query results. The system offers a flexible way to refine the query by drilling down in the hierarchical structured categories. The user can explore the results in one category with the so-called List Browser or in two categories at the same time with the so-called Matrix Browser. A familiar and well-known interactive tree widget is used for the presentation of the categories and located hyperlinks, so the handling of the system is very intuitive.

Ontobroker [105] uses an hyperbolic browser to view semantic relationships between frames in F-Logic. Ontorama [93] derives by Ontobroker Java source code, but the code has been substantially re-written since 1999 and its actual version has been developed within the WEBKB-2 project [116].

The trend in the field of visualization of ontologies is quite clear: techniques exploiting information visualization issues are increasingly adopted. That affects not only the browsing of the data and metadata but also the activities of building, comparing, and integrating different ontologies.

We can argue that in the next three years such approaches will be validated against real test beds and users, providing a more robust and mature reference scenario.

3.9 Temporal Languages

In this section, we review the common features of temporally extended ontology languages developed to abstract the temporal aspects of information. Without loss of generality, we will specifically refer to a general

temporal Entity Relationship (ER) model, and we will consider that it captures the relevant common features of the models introduced by the systems TimeER [47], ERT[113], and MADS [109]. These models cover the full spectrum of temporal constructs (see [46] for an extensive overview of temporally extended ER models). We refer to ER models because they are the most mature field in temporal conceptual modeling. We use a unifying notation for temporal constructs that will capture the commonalities among the different models [2][3]. As a main characteristic, an ontology language for time-varying information should support the notion of valid time which is the time when a property holds, i.e. it is true in the representation of the world. Possibly, the ontology language should also support transaction time which records the history of database states rather than the world history, i.e. it is the time when a fact is current in the database and can be retrieved. Temporal support is achieved by giving temporal meaning to each standard nontemporal conceptual construct and then adding new temporal constructs. The ability to add temporal constructs on top of a temporally implicit language has the advantage of preserving the nontemporal semantics of conventional (legacy) conceptual schemas when embedded into temporal schemas called upward compatibility. The possibility of capturing the meaning of both legacy and temporal schemas is crucial in modeling data warehouses or federated databases, where sources may be collections of both temporal and legacy databases. Orthogonality [109] is another desirable principle: temporal constructs should be specified separately and independently for entities, relationships, and attributes. Depending on application requirements, the temporal support must be decided by the designer. Furthermore, snapshot reducibility [111] of a schema says that snapshots of the database described by a temporal schema are the same as the database described by the same schema, where all temporal constructs are eliminated and the schema is interpreted atemporally. Temporal marks are usually added to capture the temporal behavior of the different components of a conceptual schema. In particular, entities, relationships, and attributes can be either s-marked, in which case they are considered snapshot8 constructs (i.e. each of their instances has a global lifetime, as in the case they belong to a legacy diagram), vt-marked, and they are considered temporary constructs (i.e. each of their instances has a temporary existence), or unmarked, i.e. without any temporal mark, in which case they have temporally unconstrained instances (i.e. their instances can have either a global or a temporary existence). Participation constraints are distinguished between snapshot participation constraints true at each point in time and represented by a pair of values in parentheses and lifespan participation constraints evaluated during the entire existence of the entity and represented by a pair of values in square brackets[47][109][112]. Dynamic relationships between entities [109] can be either transition or generation relationships. In a transition relationship, the instances of an entity may eventually become instances of another entity. The instances of the source entity are said to migrate into the target entity, and the phenomenon is called object migration. In temporal conceptual modeling literature, two types of transitions have been considered [54][53][109]: dynamic evolution when objects cease to be instances of the source entity, and dynamic extension, otherwise. In general, type constraints require that both the source and the target entity belong to the same generalization hierarchy. Generation relationships involve different instances differently from the transition case: an instance (or set of instances) from a source entity is (are) transformed in an instance (or set of instances) of the target entity. We have considered, so far, a temporally enhanced ontology language able to represent time varying data. Now, we want to introduce a conceptual data model enriched with schema change operators that can represent the explicit evolution of the schema while maintaining a consistent view of (static) instantiated data. The problem of schema evolution becomes relevant in the context of long-lived database applications, where stored data are considered worth surviving changes in the database schema [103]. One of the fundamental issues in the introduction of schema change operators in an ontology language is the semantics of change, which refers to the effects of the change on the schema itself, and, in particular, checking and maintaining schema consistency after changes. In the literature, the problem has been widely studied in relational and object-oriented database papers. In the relational field [28][102], the problem is solved by specifying a precise semantics of the schema changes, for example, via algebraic operations on catalogue and base tables. However, the related consistency problems have not been considered so far. The correct use of schema changes is completely under the database designer/administrator s responsibility without automatic system aid and control. In the object-oriented field, two main approaches were followed to ensure consistency in pursuing the semantics of change problem. The first approach is based on the adoption of invariants and rules and has been used, for instance, in the ORION [9] and O2 [37] systems. The second approach, which was proposed in [97], is based on the introduction of axioms. In the former approach, the invariants define the consistency of a schema, and definite rules must be followed to maintain the invariants satisfied after each schema change. Invariants and rules are strictly dependent on the underlying object model, as they refer to specific model elements. In the latter approach, a sound and complete set of axioms (provided with an inference mechanism) formalizes dynamic schema evolution, which is the actual management of schema changes in a system in operation. The approach is general enough to capture the behavior of several different systems and, thus, it is useful for comparing them in a unified framework. The

[8] See the consensus glossary [10] for the terminology used.

compliance of the available primitive schema changes with the axioms automatically ensures schema consistency without the need for explicit checking, as incorrect schema versions cannot actually be generated.

4. REASONING

We use the term reasoning over an ontology to mean any mechanism/procedure for making explicit a set of facts that are implicit in an ontology. There are many reasons why one would like to have well-founded methods for reasoning about ontologies. Here, we would like to single out two important purposes of reasoning:

- *Validation*. Validating an ontology means ensuring that the ontology is a good representation of the domain of discourse that you aim at modeling. Reasoning is extremely important for validation. For example, checking whether your ontology is internally consistent is crucial: obviously, no inconsistent theory can be a good representation of a domain of discourse. But, in turn, consistency checking is a kind of reasoning, actually the most basic one.
- *Analysis*. While in validation one looks at the ontology and tries to answer the question of whether the ontology correctly models the domain, in analysis one assumes that the ontology is a faithful representation of the domain, and tries to deduce facts about the domain by reasoning over the ontology. In a sense, we are trying to collect new information about the domain by exploiting the ontology. Obviously, analysis can also provide input to the validation phase.

The basic problem to address when talking about reasoning is to establish which is a correct way to single out what is implicit in an ontology. But what is implicit in an ontology strongly depends on the semantics of the language used to express the ontology itself. It follows that it makes no sense to talk about reasoning without referring to the formal semantics of the language used to express the ontology.

We remind the reader that in Section 2 we introduced a classification of languages based on three criteria, where the third criterion was the one taking into account the degree in which the various languages deal with the representation of incomplete information in the description of the domain. More precisely, under this criterion, we considered two classes of languages, called single model and multiple models, respectively. Now, it is immediate to verify that the world reasoning as used in logic usually refers to the task of deriving implicit knowledge in the case of multiple models. On the other hand, when we deal with ontologies interpreted in one model, deriving implicit information is a form of computation over the single model.

Based on the above considerations, in the rest of this section we will deal with the two forms of reasoning in two separate subsections.

- In subsection 4.1, we address the issue of logical reasoning, i.e., the issue of deriving implicit information in the case where the ontology language has the expressive power to state incomplete information over the domain. As an example, suppose our ontology sanctions that every male is a person, every female is a person, every person has a city of birth, and x is either a male or a female. There are two logical models of this ontology: in one model x is a male, and therefore a person, and therefore has a city of birth, in the other model x is a female, and therefore a person, and therefore has a city of birth. In both cases, one concludes that x has a city of birth. Note that is an instance of pure logical reasoning: what is true in every model of the ontology is a derived conclusion of the ontology.
- In subsection 4.2, we briefly mention the issue of computing properties that hold in an ontology characterized by a single model. As an example, suppose our ontology is expressed in the form of a relational database. Every query expressed over the database computes a set of tuples, according to the property expressed by the query. Such tuples can be seen as a kind of knowledge that is implicit in the database. However, their elicitation is not obtained by means of a process looking at several models of the ontology. Rather, the tuples are made explicit by means of a computation over the ontology, in particular the one that computes the answers to the query.

4.1 Logical reasoning

Logical reasoning has been the subject of intensive research work in the last decades. Here, we are especially interested in automated reasoning, i.e., in the task of automating reasoning. While this was already the goal of several philosophers in the past centuries, interest in automated reasoning has grown with the birth of Artificial Intelligence and Knowledge Representation.

There are several forms of logical reasoning.

- *Reasoning in classical logic*. The basic one is related to the idea of proving theorems in classical first order logic. This can be characterized as the task of checking whether a certain fact is true in every models of a first-order theory.
- *Reasoning in non-classical logics*. Although reasoning in classical logic is of paramount importance, it is not the only form of reasoning we are interested in when building ontologies. A famous example of reasoning that cannot be captured by classical logic is the so-called defeasible reasoning. Although we can state in an ontology that all birds fly, it is known that penguins are birds that do not fly. This is a form of non-monotonic reasoning: new facts may invalidate old conclusions. Since classical logic is monotone, this goes beyond the expressive power of classical logics.

No matter of whether we are performing reasoning within a classical or a non-classical logic, we can classify reasoning starting from another perspective, namely, the type of desired conclusions we are aiming at. According to this classification, we distinguish among the following:

- *Deduction*. Let W and s be the intensional level and the extensional level of an ontology, respectively. P is a deductive conclusion from W if it holds in every situation (extensional level) coherent with W and s. Examples of classical deductive logical reasoning:
  - Check whether a concept is inconsistent, i.e., if it denotes the empty set in all the models of the ontology,
  - Check whether a concept is a subconcept of another concept, i.e., if the former denotes a subset of the set denoted by the latter in all the models of the ontology
  - Check whether two concepts are equivalent, i.e., if they denote the same set in all models of the ontology
  - Check whether an individual is an instance of a concept, i.e., if it belongs to its extension in all the models of the ontology
- *Induction*. Let W and s be the intensional level and the extensional level of an ontology, respectively. Let t be a set of observations at the extensional level. We say that P is an inductive conclusion with respect to W,s and t if P is an intensional level property such that
  - W,s do not already imply t
  - P is consistent with W,s and t
  - W,s,P imply t

  Induction can be used, for example, to single out new important concepts/class in an ontology, or to come up with new axioms regarding such concepts, based on observations on individuals. In this sense, induction is the basic reasoning for learning.
- *Abduction*. Let W and s be the intensional level and the extensional level of an ontology, respectively. Let t be a set of observations at the extensional level. We say that E is an abductive conclusion wrt W,s and t if E is an extensional level property such that
  - W,s do not already imply t
  -