A Review of Ontology and Web Service Composition

Ji Zhang
CSIRO Tasmanian ICT Centre
Hobart, TAS
August 2008
Email: Ji.Zhang@CSIRO.au

1. Ontology

1.1 Definition of Ontology

The term ontology is originated from Philosophy and was adopted by AI researchers to describe formal domain knowledge. Several ontology definitions have been proposed in the last decades. The most frequently cited definition is that given by Gruber in 1993, that is, ontology is defined as "*an explicit specification of a conceptualization*". In other words, an ontology is a domain model (conceptualization) which is explicitly described (specified). Borst defines ontology as a "*formal specification of a shared conceptualization*" later [Borst, 1997]. This definition requires, in addition to Gruber's definition, that the conceptualization should express a shared view between several parties, a consensus rather than an individual view. Also, this conceptualization should be expressed in a machine readable format. In 1998, Studer et al. [Studer et al., 1998] merge these two definitions as "*An ontology is a formal, explicit specification of a shared conceptualization*."

Classes are the major constituting components of most ontologies. Classes are the entities describing concepts in the domain under study. For example, a class of wines represents all wines. Specific wines are instances of this class. A class may have several *subclasses* that represent concepts that are more specific than their corresponding superclass. For instance, the class of all wines can be further divided into red, white, and rosé wines. In this case, red, while and rose wines are all subclasses of the class of wine. Besides classes of concepts, an ontology can defines the properties of each class of concepts and the constraints the concepts are subject to. In a word, ontology is a formal and explicit description of concepts in a domain (classes), with properties of each concept describing various features (slots), and restrictions on slots (facets). An ontology together with a set of individual instances of classes constitutes a *knowledge base*.

1.2 The Usefulness of Ontology

The reasons for developing and using ontology can be summarized as follows:

- **To share common understanding of the structure of information among people or software agents**

Sharing common understanding of the structure of information among people or software agents is one of common goals in developing ontologies [Musen 1992; Gruber 1993]. For example, a few web sites or web services are available in a particular domain and they all share the same underlying ontology. Based on this commonly shared ontology, compute agents are able to interoperate the information from different web sites or web services, and automatic aggregation of information is possible to answer complex user queries that involve the information from different web sites and services.

- **To enable reuse of domain knowledge**

Developing ontology greatly enables its reusability. This is one of the driving forces in ontology research. For example, models for many different domains need to represent the notion of time. This representation includes the notions of time intervals, points in time, relative measures of time, and so on. If one group of researchers develops such an ontology in detail, others can simply reuse it for their domains. Additionally, if we need to build a large ontology, we can integrate several existing ontologies describing part of the large domain. We can also reuse a general ontology, such as the Gene Ontology, and extend it to describe our domain of interest.

- **To make explicit domain assumptions**

Making explicit domain assumptions underlying an implementation makes it convenient to adjust these assumptions when the underlying knowledge about the domain changes. The way of hard-coding and embedding domain assumptions in programming-language code impedes a good understanding and an easy change of these assumptions. In addition, explicit specifications of domain knowledge are useful for new users to learn the meaning of the terms defined in the domain.

- **To separate domain knowledge from the operational knowledge**

By using ontologies, we are able to better separate the domain knowledge from the operational knowledge. For example, we can describe a task of assembling a product from its components and develop a program that does this assembling that is independent of the products and components themselves. An ontology in the domain of car can be devised to assemble cars using the program, and similarly, an ontology in the domain of boat can help assemble boats by means of this program as well.

- **To analyze domain knowledge**

Analysing domain knowledge is possible once a declarative specification of the terms is available. Formal analysis of terms is extremely valuable when both attempting to reuse existing ontologies and extending them [McGuinness et al. 2000].

Often an ontology of the domain is not a goal in itself [Noy_2]. Developing an ontology is similar in spirit to defining a set of data and their structure for other programs to use. Problem-solving methods, domain-independent applications, and software agents use ontologies and knowledge bases built from ontologies as data.

1. 3 Ontology Development Methodology

Development of an ontology can take the following generic steps:

Step 1. Determine the domain and scope of the ontology

The ontology development starts with the development work of an ontology by defining its *domain* and *scope*. The domain and scope can be formulated by simply answering the following questions:

- What is the domain that the ontology will cover?
- For what purpose we are going to use the ontology?
- For what types of questions the information in the ontology should provide answers?
- Who will use and maintain the ontology?

A good way for testing whether the scope of ontologies has been properly defined is the so-called *competency questions test*. This test involves sketching a list of questions that a knowledge base based on the ontology should be able to answer [Gruninger and Fox 1995]. The competency questions aim to evaluate the ontology to see whether it contains enough information, and in sufficient detailed level, to cover the required domain for the domain applications.

An example of competency questions

In the food domain, the following are the possible competency questions for developing the ontology of food and wine from [Noy_2]:

- Which characteristics should I consider when choosing a wine?
- Is Bordeaux a red or white wine?
- Does Cabernet Sauvignon go well with seafood?
- What is the best choice of wine for grilled meat?
- Which characteristics of a wine affect its appropriateness for a dish?
- Does a bouquet or body of a specific wine change with vintage year?
- What were good vintages for Napa Zinfandel?

Step 2. Consider the possibility of reusing existing ontologies

Before starting to develop the ontology from scratch, it is worthwhile considering the existing related ontologies that can potentially be used or become part of the ontology that is to be developed. If there are some existing ontologies that can be used, then significant development time can be saved as a result. The existing ontologies, regardless of their exact representation languages, can be translated and imported through most ontology development tools or environments, base on which the ontology to be development can be built.

Step 3. Enumerate important terms in the ontology

At the beginning of the design process, it is necessary and useful to compile a list of terms, corresponding to the concepts in the domain, that will be represented in the ontology. A

general guidance is that try to get a comprehensive list of terms without worrying too much about overlap between concepts they represent, relations among the terms, or any properties that the concepts may have, or whether the concepts are classes or slots. The subsequent steps will sort out the classes and their properties.

Step 4. Define the classes and class hierarchy

There does not exist a uniform approach for developing a class hierarchy. Instead, there are several possible approaches for doing this [Uschold and Gruninger 1996]. They are top-down approach, bottom-up approach and middle-out approach.

- **Top-down approach**

  A top-down approach starts with the definition of the most general concepts in the domain, followed by the subsequent specialization of the concepts.

- **Bottom-up approach**

  A bottom-up development process starts with the definition of the most specific classes, the leaves of the hierarchy, with subsequent grouping of these classes into more general concepts.

- **Middle-out approach**

  A middle-out approach is a combination of the top-down and bottom-up approaches: We define the more salient or important concepts first and then generalize and specialize them appropriately at the same time.

Step 5. Define the properties of classes (slots)

Once we have defined the classes and its hierarchy, we then can start to design the property (or called slots) associated with each concept for describing various characteristics of the concepts.

Step 6. Define the constraints of slots (facets)

Besides using properties to describe concepts, we also need to specify the constraints (called facets) the concept and properties should satisfy. Facets can be diverse and are related to the value type, allowed values, the number of the values (cardinality), and other features of the values the slot can take. We briefly discuss two major facets, *slot cardinality* and *slot value-type*, as follows:

- **Slot cardinality**

  Slot cardinality defines how many values a slot is allowed to have. Some systems distinguish only between single cardinality (allowing at most one value) and multiple cardinality (allowing any number of values). Some systems allow specification of a minimum and maximum cardinality to describe the number of slot values more precisely. Minimum cardinality of *N* means that a slot must have at least *N* values.

- **Slot value-type**

A value-type facet describes what types of values can a slot can take. The commonly used value-type for slots includes string, number (float/integer), Boolean and enumerated.

- **String** is the simplest value type which is used for slots such as name;
- **Number** (sometimes more specific value types of Float and Integer are used) describes slots with numeric values;
- **Boolean** slots are simple “yes”/”no” label indicate the binary nature of the slots;
- **Enumerated** slots specify a list of specific allowed values for the slot.

Step 7. Create instances

The last step is creating individual instances of classes in the hierarchy. This step is called *ontology instantiation*. Defining an individual instance of a class requires (1) choosing a class, (2) creating an individual instance of that class, and (3) filling in the slot values.

1. 4 Ontology Representation Languages

For ontologies to be used within an application, the ontology must be specified and represented in a certain language. This refers to the encoding of ontology. There are a variety of formal languages which can be used for representation of conceptual models, with varying characteristics in terms of their expressiveness, ease of use and computational complexity. The field of knowledge representation (KR) has, of course, long been a focal point of research in the Artificial Intelligence community [Duce1988]. The major considerations in the choice of representation language for ontologies are the expressivity, the rigour and the semantics of a language, which are discussed as follows.

- **Language expressivity**

  The expressivity of a representation language is a measure of the range of constructs that can be use to formally, flexibly, explicitly and accurately describe the components of an ontology. For example, first order logic is a very expressive representation language. Yet, there is a trade-off between expressivity (what you can say) and complexity (whether the language is computable in real time).

- **Language rigor**

  The rigour of a language is a measure of the satisfiability and consistency of the representation within the ontology. A model is satisfiable if none of the statements within contradict each other. Language is desired to achieve consistency within an ontology that encodes the conceptualisation of the knowledge in the same manner throughout the ontology. The rigour of an ontology's representational scheme should be maintained by the systematic enforcement of mechanisms using the ontology, which ensures the uniform and universal interpretation of the ontology.

- **Language semantics**

  The semantics of a language refers to the ability of the language in unambiguously present the meaning it wants to express. Clearly defined and well-understood semantics

are essential. The definition of a general exchange language for ontologies is the subject of much current effort in the ontology research community [Horrocks]. In terms of semantics, languages currently used for specifying ontologies fall into three kinds: vocabulary representation; object-based knowledge representation languages such as frames and UML, and languages based on logic such as Description Logics.

- **Vocabulary representation**

  Vocabulary-like representation of ontology supports the purely hand-crafted ontologies with simple tree-like structures with explicitly defined inheritance. The example of ontology with vocabulary representation is the Gene Ontology that features a hierarchical structure of gene products. The location of each concept and its relation with others in the ontology is completely determined by the modeller. Each entry or concept in the GO has a name, an identifier and other optional pieces of information such as synonyms, references to external databases and so on. Although this provides great flexibility, the lack of any structure in the representation can lead to difficulties with maintenance or preserving consistency, and there are usually no formally defined semantics. The single inheritance provided by a tree structure (each concept has only one parent in the is-a hierarchy) can also prove limiting.

- **Frame-based representation**

  The frame-based representation is based upon the concept of frames (or classes) which represent collections of instances in the ontology. Each frame has an associated collection of slots or attributes which can be filled by values or other frames. In particular, frames can have a kind-of slot which allows the representation of a frame taxonomy. A frame provides a context for modelling one aspect of a domain. An important part of frame-based languages is the possibility of inheritance between frames. The inheritance allows inheriting attributes together with restrictions on them. Knowledge base then consists from instances (objects) of these frames. Frame-based systems have been used extensively in the KR domain, particularly for applications in natural language processing. The most well known frame system is Ontolingua [Farquhar1997]. Frames are very popular because frame-based modelling is similar to object-based modelling and is intuitive for many users.

- **Logical representation**

  An alternative to frames is logic representation, especially Description Logics (DLs) [Woods1992, Bechhofer1999]. Description logic was designed as an extension to frames, which are not equipped with formal logic-based semantics. DLs describe knowledge in terms of concepts and relations that are used to automatically derive classification taxonomies. A major characteristic of a DL is that concepts are defined in terms of descriptions using other properties and concepts. For instance, the concept Enzyme was not simply asserted by the modellers in the ontology. Instead, a composite concept was made from Protein and Reaction that are linked with the relation catalyses. Enzyme is thus defined as the Protein which catalyses Reaction. In this way, the model is built up from small pieces in a descriptive way, rather than

through the assertion of hierarchies. The DL provides a number of reasoning services. Applications can draw on these reasoning services to make use of the knowledge represented in the ontology [Bechhofer1999].

1.5 Ontology Development/Editing Tools

There have been more than 90 tools available for ontology development (ontology viewing and editing) from both non-commercial organizations and commercial software companies. Most of them are tools for designing and editing ontologies. Some of them may provide in addition certain capabilities for analyzing, modifying, and maintaining ontology evolution over time.

The representatives of the ontology development/editing environments are Protégé 2000, WebODE and OntoEdit. A common characteristics shared by these environments are that they are built as robust integrated environments or suites that provide technological support to most of the ontology lifecycle activities. They have extensible, component-based architectures, where new modules can easily be plugged-in to provide more functionality to the environment. Besides, the knowledge models for ontologies underlying these environments are language independent. A wide spectrum of representation languages are support by these tools through explicit language importation and exportation facilities.

- **Protégé 2000**

Protégé 2000 [Noy 2000] was developed by the Stanford Medical Informatics (SMI) at Stanford University. Protégé 2000 is an open source, standalone integrated software tool used by system developers and domain experts to develop knowledge-based systems. The core of this Protégé 2000 is the ontology editor, and it provides a library of plugins that add more functionality to the environment. Currently, plugins are available for ontology language importation/exportation (FLogic, Jess, OIL, XML, Prolog), constraints creation and execution (PAL), ontology merge [Noy_1 2000], etc.

- **WebODE**

WebODE [Arpırez2001] was developed in the Artificial Intelligence Lab from the Technical University of Madrid (UPM). It is also an ontology-engineering suite created with an extensible architecture. WebODE is not used as a standalone application, but as a Web server with a Web interface. The core of this environment is the ontology access service, which is used by all the services and applications plugged into the server, especially by the WebODE Ontology Editor. There are several services for ontology language importation/exportation (XML, RDF(S), OIL, DAML+OIL, CARIN, FLogic, Jess, Prolog), axiom edition with WebODE Axiom Builder (WAB) [Corcho2002], ontology documentation, ontology evaluation and ontology merge.

- **OntoEdit**

OntoEdit [Sure2002] was developed by in Karlsruhe University, Germany. It is an extensible and flexible environment, based on a plugin architecture, which provides functionality to

browse and edit ontologies. It includes plugins for inferring using Ontobroker [Decker1999], of exporting and importing ontologies in different formats (FLogic, XML, RDF(S), DAML+OIL), etc. Two versions of OntoEdit are available: OntoEdit Free and OntoEdit Professional. The successor of OntoEdit, called KAON (Karlsruhe Ontology) tool suite has been developed.

- **Ontology Management Systems**

An ontology management system for ontology is equivalent to a database management system (DBMS) for data. A DBMS allows an application to externalize the storing and processing of data, via a standard interface, and relieves the program from the burden of deciding how to store the data in files, how to index the data, how to optimized queries, how to retrieve query results, etc. Likewise, an ontology management system allows an application to manipulate and query ontology without one having to worry about how the ontology is stored and accessed, how queries are processed, how query results are retrieved, etc., by providing a programming interface. Ontology editing capabilities are not the critical component of an ontology management system. An ontology management system may or may not provide the ontology editing and designing capabilities. In case it does not, an ontology management system can be used together with an ontology editor such as Protégé, if necessary.

Snobase Ontology Management System [Snobase], developed by IBM, provides mechanisms for loading ontologies from files and via the Internet, and for locally creating, modifying, and storing ontologies. It also provides a mechanism for querying ontologies, as well as an easy-to-use programming interface so that applications can interact in standard ontology specification languages such as RDF, DAML+OIL, and OWL. Internally, the system uses an inference engine, an ontology persistent store, an ontology directory, and ontology source connectors. Applications can query against the created ontology models and the inference engine deduces the answers and returns results sets.

1. 6 Life-cycle of Ontology Development

Similar to software development, there is also a life-cycle of ontology development include a set of stages that occur when building ontologies, guidelines and principles to assist in the different stages. Figure 1 presents the popular V-model life-cycle for ontology development. Various stages in this model are explained below.

1. Identify purpose and scope. Come up with a requirements specification for the ontology by identifying the intended scope and purpose of the ontology. A well-characterized requirements specification is important to the design, evaluation and re-use of an ontology.

2. Knowledge Acquisition. Knowledge acquisition refers to the process of acquiring necessary domain knowledge to build the ontology. Knowledge may come from a diverse of sources for a particular domain. For example, in the biology domain, the knowledge may come from biologists, database metadata, standard text books, research papers and other existing ontologies. Motivating scenarios are collected and informal competency questions

formed [Uschold1996]. These are questions that the ontology is desired to answer and will be used to check the applicability of ontology.

3. Conceptualization. Conceptualization involves identifying the key concepts that exist in the domain, their properties and the relationships that hold between them; identifying natural language terms to refer to such concepts, relations and attributes; and structuring domain knowledge into explicit conceptual models.

4. Encoding. Encoding is the implementation process for materializing the conceptualization using some formal language for ontology representation, e.g. vocabulary, frames or logic. This process can be greatly streamlined by using ontology development tools or environment that provides a rich set of ontology construction and visualization facilities.

5. Documentation: Documentation is an important work after ontology has been implemented. Documentation may record the information regarding the informal and formal complete definitions, assumptions and examples to promote the appropriate use and the later re-use of an ontology.

6. Evaluation: Evaluation of ontology helps assess the appropriateness of an ontology for its intended application. Evaluation is done pragmatically, by assessing the competency of the ontology to satisfy the requirements of its application, including determining the consistency, completeness and conciseness of an ontology [Gomez-Perez1994].

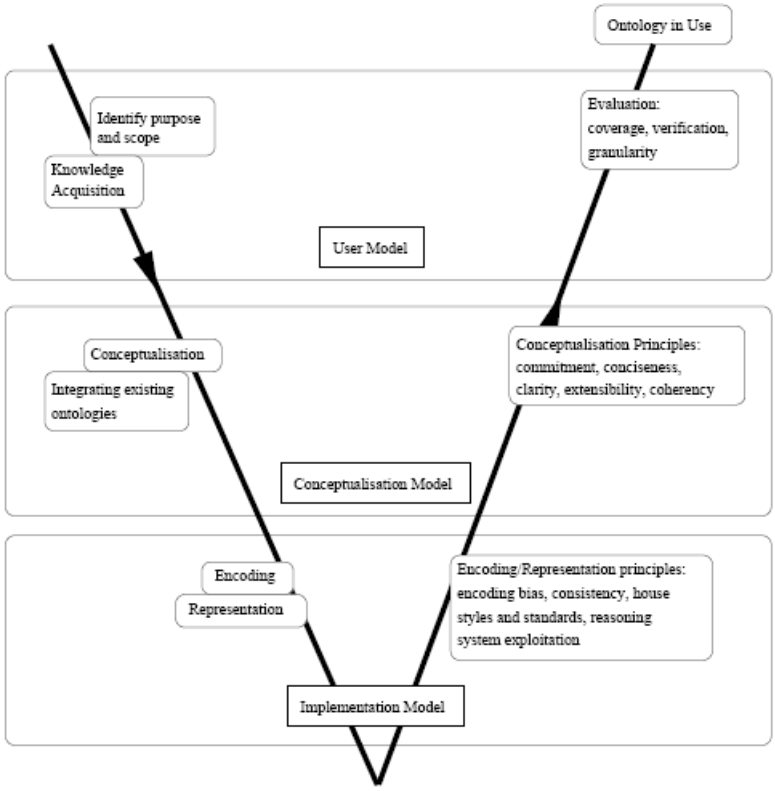

Figure 1. The V-model inspired methodology for building ontologies

1. 7 Operations on Ontologies

It is possible that one application uses multiple ontologies, especially when using modular design of ontologies or when we need to integrate with systems that use other ontologies. In this case, some operations on ontologies may be needed in order to work with all of them.

- **Merge**

Merge of ontologies means creation of a new ontology by linking up the existing ones. Conventional requirement is that the new ontology contains all the knowledge from the original ontologies, however, this requirement does not have to be fully satisfied, since the original ontologies may not be together totally consistent. In that case the new ontology imports selected knowledge from the original ontologies so that the result is consistent. The merged ontology may introduce new concepts and relations that serve as a bridge between terms from the original ontologies.

- **Mapping**

Mapping from one ontology to another one is expressing of the way how to translate statements from ontology to the other one. Often it means translation between concepts and relations. In the simplest case it is mapping from one concept of the first ontology to one concept of the second ontology. It is not always possible to do such one to one mapping. Some information can be lost in the mapping. This is allowed, however the bottle-line is that mapping may not introduce any inconsistencies.

- **Alignment**

Alignment is a process of mapping between ontologies by modifying or re-organizing original ontologies so that suitable translation exists (i.e., without losing information during mapping). Thus it is possible to add new concepts and relations to ontologies that would form suitable equivalents for mapping.

1. 8 Major Domain-Ontology in Biology: Bio-Ontologies

The use of ontology within bioinformatics is relatively recent. In this section, a representative sample of existing bio-ontologies will be reviewed. This list is limited to those ontologies most relevant to current trends in bioinformatics and molecular biology, rather than the wider field of biology.

- **The Ontology of Genes--**Gene Ontology (GO) [GO]

The most prominent ontology for bioinformatics is Gene Ontology (GO). GO project is a collaborative effort to address the need for consistent descriptions of gene products in different databases. The GO project has developed three structured controlled vocabularies (ontologies) that describe gene products in terms of their associated biological processes, cellular components and molecular functions in a species-independent manner.

- **Pathway Ontology and Databases**

- **BioPAX** [BioPAX]

  BioPAX is a collaborative effort to create a data exchange format for biological pathway data. BioPAX has two levels of formats. BioPAX level-1 represents metabolic pathway information and Level 2 represent signaling, genetic regulatory and genetic pathways. BioPAX is anOWL ontology.

- **KEGG** [KEGG]

  The Kyoto Encyclopedia of Genes and Genomes is the primary database resource of the Japanese Genome Net service for understanding higher-order functional meanings and utilities of the cell or the organism from its genome information. KEGG consists of the PATHWAY database for the computerized knowledge of molecular interaction networks such as pathways and complexes.

- **EcoCyc**[Ecocyc]

  EcoCyc is an organism-specific pathway database that describes the metabolic and signal transduction pathways. EcoCyc and MetaCyc are part of the BioCyc relational database, which is available as a collection of flat files.

- **MetaCyc** [MetaCyc]

  MetaCyc is a metabolic-pathway database that describes non-redundant elucidated metabolic pathways from more than 240 different organisms. Applications of MetaCyc include pathway analysis of genomes, metabolic engineering and biochemistry education. Metacyc and EcoCyc can be queried using the Pathway tools GUI, which provides a wide variety of query operations and visualization tools.

- **Ontology for Microarray experiments- MGED Ontology**

MGED Ontology is part of a coordinated effort by members of the Microarray community through the Microarray Gene Expression Data (MGED) Society. MGED [MGED] has generated a set of guidelines for supplying the minimal information about a Microarray experiment (MIAME, [Brazma2001]). These guidelines have provided the foundation for concepts to be included in the ontology. A foundation for the relationships between the concepts was provided by the MAGE effort [Spellman2002], developed jointly by MGED, Rosetta, and others. MAGE (Microarray gene expression) is an object model that has been formally accepted as a standard by the Object Management Group (OMG, [OMG]) and implemented as a form of XML (MAGE-ML).

The terms for organism part used in the Microarray experiments are not provided by either MIAME or MAGE, neither are terms for describing the age of the sample, the experimental design or the types of protocols used.

The MGED ontology is an ontology for describing or annotating experiments, specifically Microarray experiments, but potentially extensible to other types of functional genomics experiments. Although the major component of the ontology involves biological descriptions, it is not an ontology of molecular, cellular or organism biology. Rather, it is an ontology that

includes concepts of biological features relevant to the interpretation and analysis of an experiment.

- **Other Bio-ontologies**
  - **The TAMBIS Ontology** [TaO]

    TAMBIS (Transparent Access to Multiple Bioinformatics Information Sources) uses an ontology to enable biologists to ask questions over multiple external databases using a common query interface. The TAMBIS ontology (TaO) [TaO] describes a wide range of bioinformatics tasks and resources, and has a central role within the TAMBIS system. An interesting difference between the TaO and some of the other ontologies, is that the TaO does not contain any instances. The TaO only contains knowledge about bioinformatics and molecular biology concepts and their relationships - the instances they represent still reside in the external databases. The TaO is available in two forms - a small model that concentrates on proteins and a larger scale model that includes nucleic acids.

  - **The RiboWeb Ontology** [RiboWeb]

    A knowledge base containing an ontology-based representation of the primary data relevant to the structure of the ribosome as well as supplementary functional data. In particular, the main experimental results of approximately 200 articles that are key for ribosomal structure modelling have been coded in this knowledge base.

2. Semantic Web Services

2.1 Web Service Architecture and Enabling Techniques

The architecture for Web services, as shown in Figure 2, can be defined by three phases; publish, discovery and bind. Three entities are involved: the service requester, which invokes services; the service provider which responds to requests; and the registry where services can be published or advertised. A service provider publishes a description of a service it provides to a service registry. This description (or advertisement) includes a profile on the provider of the service (e.g. company name and address); a profile about the service itself (e.g. name, category); and the URL of its service interface definition (i.e. WSDL description).

The enabling techniques for Web service are shown in Figure 3. A Web service interface is described using the Web Service Description Language 1.1 (WSDL). Web services exchange messages encoded in the SOAP (Simple Object Access Protocol) messaging framework and transported over HTTP or other Internet protocols. Several tasks can be performed with Web services. A typical Web service usage life-cycle can be described as follows. A service provider publishes the WSDL description of his service in UDDI, a registry that permits Universal Description Discovery and Integration of Web services. Subsequently, service requesters can search against UDDI and locate/discover Web services that are of interest. Using the information provided by the WSDL description they can directly invoke the corresponding Web service. Further, several Web services can be composed to achieve a

more complex functionality. Such compositions of services can be specified using BPEL4WS14 (Business Process Execution Language for Web Services).

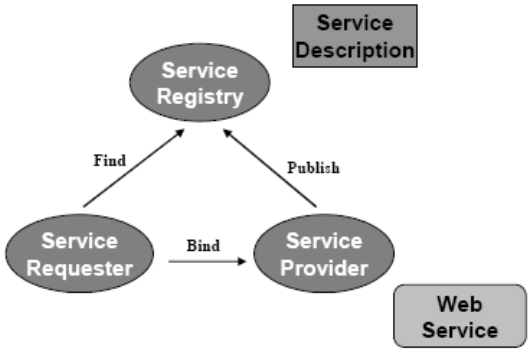

Figure 2. Web service architecture

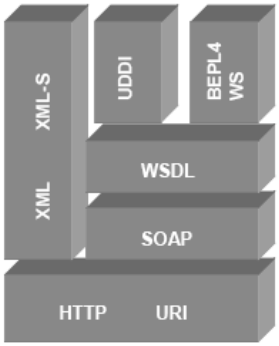

Figure 3. Web service standards

- **SOAP**

Today's applications communicate using Remote Procedure Calls (RPC) between objects like DCOM and CORBA, but HTTP was not designed for this purpose. RPC suffers a compatibility and security problem and are normally blocked by firewalls and proxy servers. A better way to communicate between applications is over HTTP, because HTTP is supported by all Internet browsers and servers. SOAP was created on top of HTTP to accomplish this.

SOAP provides a basic messaging framework for Web services to exchange messages. SOAP defines an XML-based format for the specification of structured and typed messages that can be exchanged by Web services (through the so-called SOAP endpoints) in a distributed environment. In addition, SOAP provides a way to describe the actions that an endpoint must take on receiving a particular SOAP message. SOAP is serving as a basic building block for Web services and is therefore a common protocol used for several Web services standards. It is tightly used with WSDL.

- **UDDI**

UDDI (Universal Description, Discovery and Integration) is an initiative to develop a standard for an online registry, to enable the publishing and dynamic discovery of Web services offered by businesses [UDDI]. UDDI is an open industry initiative, sponsored by Organization for the Advancement of Structured Information Standards (OASIS) [OASIS].

In UDDI, each business is described using a *businessEntity* element that describes a business by name, a key value, categorisation, services offered (businessService elements) and contact information for the business. A *businessService* element describes a service using a name, key value, categorisation and multiple bindingTemplate elements (analogous to a Yellow Pages element that categorises a business). A *bindingTemplate* element describes the kind of access the service requires (phone, mailto, http, ftp, fax etc.), key values and tModelInstances. *tModelInstances* are used to describe the protocols, interchange formats that the service comprehends, that is, the technical information required to access the service.

UDDI provides a venue for registering Web services. A Web service provider registers its advertisements for services using keywords for categorisation. A Web services user retrieves advertisements out of the registry based on keyword search. So far, the UDDI search mechanism relied on predefined categorisation through keywords.

- **Web Service Description Language (WSDL)**

The Web Service Description Language (WSDL) [Christensen & others, 2001] is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services). Four types of communication are defined involving a service's operation (endpoint): the endpoint receives a message (one-way), sends a message (notification), the endpoint receives a message and sends a correlated message (request-response), and it sends a message and receives a correlated message (solicit-response). Operations are grouped into port types, which describe abstract end points of a Web service such as a logical address under which an operation can be invoked. A WSDL message element defines the data elements of an operation. XML Schema syntax is used to define platform-independent data types in the messages. Each message can consist of one or more parts. The parts are similar to the parameters of a function call in a traditional programming language. Concrete protocol bindings and physical address port specifications complete a Web service specification.

SOAP, WSDL, UDDI, and BPEL4WS are the standard combination of technology to build a Web service application. However, they fail to achieve the goals of automation and interoperability because they require human effort [Lassila, 2002]. Indeed, WSDL specifies the functionality of the service only at a syntactic level. While these descriptions can be automatically parsed and invoked by machines, the interpretation of their meaning is left for human users.

2.2 Semantic Web and Enabling Techniques

The Semantic Web is a Web of meaningful contents and services, which can be interpreted directly by computer programs. Semantic Web provides an efficient way of representing data on the World Wide Web, or as a globally linked database. The current key components of the Semantic Web framework, see Figure 4, are RDF, RDF Schema (RDF-S) and the Web Ontology Language (OWL). They are ontology languages have a rich set of constructs and can be used to present semantics-rich web information.

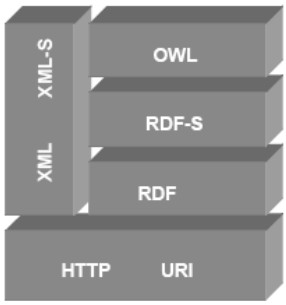

Figure 4. Semantic Web and enabling techniques

Ontologies are explicitly specified in a formal language. We require that the specification is formal (like a program is formally written in a programming language) so that the ontology can be processed by a computer. The two major ontology languages for Semantic Web are RDF(s) and OWL.

- **RDF(S)**

RDF and RDF Schema (RDF-S) are the first step towards a Web based ontology language. RDF is a data model allowing to describe resources on the Web. An RDF description is a set of triples where each triple resembles the subject, verb and object of a sentence. Each element of the triple is represented by a Universal Resource Identifier (URI). RDF can be depicted using RDF graph of data. RDF can be written in multiple notations such as XML and Notation3, etc. RDF Schema is based on RDF and allows the definition of basic ontology elements such as classes and their hierarchy, properties with their domain, range and hierarchy [McBride, 2004]. Thus, RDFS can be used to describe taxonomies of classes and properties and is well suited for expressing lightweight ontologies.

Next, we show an example of RDF and its corresponding RDF Graph (shown in Figure 5):

```
<rdf:RDF
          xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
          xmlns:foaf="http://xmlns.com/foaf/0.1/"
          xmlns="http://www.example.org/~joe/contact.rdf#">
        <foaf:Person rdf:about=
              "http://www.example.org/~joe/contact.rdf#joesmith">
          <foaf:mbox rdf:resource="mailto:joe.smith@example.org"/>
          <foaf:homepage
                    rdf:resource="http://www.example.org/~joe/"/>
          <foaf:family_name>Smith</foaf:family_name>
          <foaf:givenname>Joe</foaf:givenname>
        </foaf:Person>
      </rdf:RDF>
```

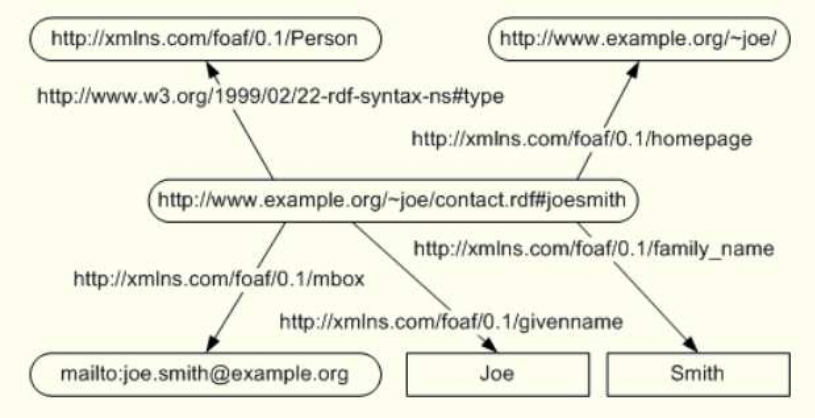

Figure 5. RDF graph example

To query RDF, SPARQL (The Simple Protocol and RDF Query Language) can be used. SPARQL is a SQL-like language for querying RDF data. For expressing RDF graphs in the matching part of the query, TURTLE syntax is used. An example of a SELECT query follows.

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
WHERE { ?x foaf:name ?name .
        ?x foaf:mbox ?mbox . }
```

The first line defines namespace prefix, the last two lines use the prefix to express a RDF graph to be matched. Identifiers beginning with question mark ? identify variables. In this query, we are looking for resource ?x participating in triples with predicates foaf:name and foaf:mbox and want the subjects of these triples.

- **Web Ontology Language (OWL)**

The OWL Web Ontology Language is a W3C recommendation for a web ontology language and is the representation language for OWL-S ontologies. The OWL is developed based on DAML+OIL language which originated by merging two language proposals that aimed at overcoming the expressivity limitations of RDF(S): DAML-ONT and OIL [Horrocks et al., 2003]. OWL enhances the expressivity of RDF(S) providing means to describe relations between classes (e.g., disjointness, union, intersection), cardinality and value restrictions on

properties (e.g., cardinality, universal and existential quantifiers), property characteristics (e.g., transitivity, symmetry), equality etc.

One major reason OWL was chosen as the representation language for OWL-S is that it is compatible with XML [Bray 2000] and RDF [Klyne2004] and at the same time provides additional expressiveness to allow users to formally describe more types of classes, properties, individuals, and relationships than XML or RDF. OWL also provides a formal semantics, thus terms defined using OWL are given a precise meaning and they can be used effectively in applications that require interoperability. OWL is a general purpose representation language that provides no special vocabulary for service applications. Thus anyone who needs to build a service application would need to find a service ontology or build their own.

OWL provides three increasingly expressive sublanguages: OWL Lite, OWL DL, and OWL Full. OWL Lite provides less expressive power than OWL DL and OWL Full and was not expressive enough for OWL-S needs. OWL DL provides maximum expressiveness while retaining computational completeness and decidability and thus is a good choice of representation language if efficient reasoning support is sought. (OWL DL and OWL Full provide the same language constructs but OWL DL places some restrictions on usage in order to avoid problems with completeness and decidability.) OWL-S ontologies are written in OWL DL.

2.3 **Semantic Web Services**

The Semantic Web community addressed the limitations of current Web service technology by augmenting the service descriptions with a semantic layer. With the Semantic Web infrastructure that becomes increasingly mature, powerful applications can be developed that use annotations and suitable inference/planning engines to automatically discover, execute, compose, and interoperate Web services.

The Semantic Web [Berners-Lee, Hendler, & Lassila, 2001] views the World Wide Web as a globally linked database where web pages or web services are marked with semantic annotations. A Semantic Web Service is defined through a service ontology, which enables machine interpretability of its capabilities as well as integration with domain knowledge. The deployment of Semantic Web Services will rely on Web Services and Semantic Web enabling technologies. There exist several initiatives (e.g. http://dip.semanticweb.org or http://www.swsi.org) taking place in industry and academia, namely OWL-S [Martin et al., 2003] and WSMO [WSMO].

All emerging frameworks for Semantic Web service descriptions (OWL-S and WSMO) is that they combine two kinds of ontologies to obtain a service description. First, a generic Web service ontology, such as OWL-S, specifies generic Web service concepts (e.g., Input, Output) and prescribes the backbone of the semantic Web service description. Second, a domain ontology specifies knowledge in the domain of the Web service, such as types of

service parameters (e.g., City) and functionalities (e.g., FindMedicalSupplier), that fills in this generic framework.

2.3.1 (Generic) Web Service Ontology

Two generic ontologies for Web service descriptions are available. First, DAML-S is an ontology that permits describing several aspects of a Web service. DAMLS was translated from DAML to OWL and renamed to OWL-S. The second, more recent, initiative is WSMO (Web Service Modelling Ontology).

- **OWL-S Ontology**

OWL-S is a set of several interrelated OWL ontologies that provide a set of well defined terms for use in service applications. The OWL-S ontologies define terms commonly used in service profiles, process models, and service groundings. The OWL-S ontologies provide semantic web users with an existing vocabulary of classes, relations, and instances for use in OWL, RDF, and XML-compatible applications.

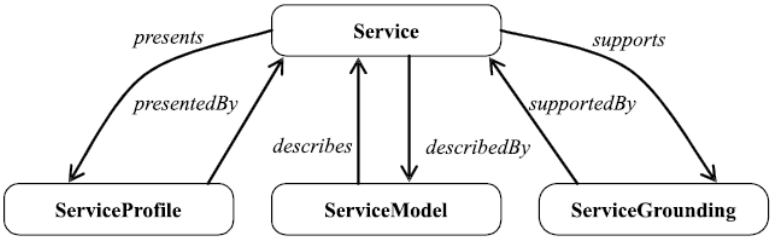

Figure 6. The OWL-S service ontology

Specifically, the OWL-S ontology is conceptually divided into four sub-ontologies for specifying what a service does, i.e. Profile [OWL-SProfile], how the service works, i.e. Process [OWL-SProcess] and how the service is implemented, i.e. Grounding [OWL-SGrounding]. A fourth ontology, i.e. Service [OWL-SService] contains the Service concept which links together a ServiceProfile, a ServiceModel and a ServiceGrounding concept (see Figure 6). The Service presents a ServiceProfile, is described by a ServiceModel and supports a ServiceGrounding. These three concepts are all further specified using the Profile, Process and Grounding ontologies respectively. Note that composition of services is not provided by OWL-S—it is a capability that must be implemented at the client level. OWL-S simply provides for the discovery of services, not their automatic composition.

- **OWL-S Profile Ontology**

  The OWL-S Profile Ontology specifies the functionality offered by the service, the semantic type of the inputs and outputs, the details of the service provider and non-functional service parameters, such as QoS parameter (e.g. service quality rating) or geographic location. The central concept of this ontology, Profile, is a subclass of ServiceProfile. The Profile is used to describe services for the purposes of discovery; service descriptions are constructed from a description of functional properties (i.e. inputs, outputs, preconditions, and effects, called **IOPE**s), and non-functional

properties (human oriented properties such as service name, etc, and parameters for defining additional meta data about the service itself, such as concept type or quality of service).

- **OWL-S Process Ontology**

  The OWL-S Process Ontology describes the composition or orchestration of one or more services in terms of their constituent processes. This is used both for reasoning about possible compositions (such as validating a possible composition, determining if a model is executable given a specific context, etc) and controlling the enactment/ invocation of a service. Three process classes have been defined: the atomic, simple and composite process. The atomic process is a single, black-box process description with exposed IOPEs. Inputs and outputs relate to data channels for data flows between processes. Preconditions specify facts of the world that must be satisfied in order to execute a service. Effects characterize the consequence of a successful execution of the service, such as the physical side-effects that the execution the service has on the physical world. Simple processes provide a means of describing service or process abstractions, such elements have no specific binding to a physical service, and thus have to be realized by an atomic process, or expanded into a composite process. Composite processes are hierarchically defined workflows, consisting of atomic, simple and other composite processes. These process workflows are constructed using a number of different composition constructs, including: Sequence, Unordered, Choice, If-then-else, Iterate, Repeat-until, Repeat-while, Split, and Split+join.

- **OWL-S Grounding Ontology**

  The OWL-S Grounding Ontology provides the vocabulary to link the conceptual description of the service, specified by the Profile and Process, to actual implementation details, such as message exchange formats and network protocols. The grounding to a WSDL description is performed according to three rules:

  R1: Each AtomicProcess corresponds to one WSDL operation;

  R2: As a consequence of the first rule, each input of an AtomicProcess is mapped to a corresponding message-part in the input message of the WSDL operation. Similarly, each output of an AtomicProcess is mapped to a corresponding message-part in the output message of the WSDL operation;

  R3:The type of each WSDL message part can be specified based on an OWL-S parameter (i.e., an XML Schema data type).

The Grounding ontology specializes the ServiceGrounding as a WSDLGrounding which contains a set of *WsdlAtomicProcessGrounding* elements, each grounding one of the atomic processes specified in the ProcessModel.

- **Promising Characteristics of OWL-S**

The OWL-S has the following promising characteristics:

1. OWL-S differentiates between the semantic and syntactic aspects of the described Web service. The Profile and Process ontologies allow for a semantic description of the Web service while the WSDL description encodes its syntactic aspects (such as the names of the operations and their parameters). The Grounding ontology provides a mapping between the semantic and the syntactic parts of a WS description;
2. OWL-S offers a core set of primitives through the ontologies to specify any type of Web service. These descriptions can be enriched with domain knowledge specified in a separate domain ontology. The core set of primitives are versatile and can be used across different domains;
3. OWL-S partitions the WS description based on three concepts. As a result a Service instance is associated with three instances each of them containing a particular aspect of the service. There are several advantages of this modular modelling. First, since the description is split up over several instances it is easy to reuse certain parts. For example, one can reuse the Profile description of a certain service. Second, service specification becomes flexible as it is possible to specify only the part that is relevant for the service (e.g., if it has no implementation one does not need the ServiceModel and the ServiceGrounding).

Table 1 presents a brief comparison between BPEL4WS and OWL-S in terms of the domain they are mainly applied to, whether they provide semantic representation or only a syntactic one, their interfaces and the approach used for service composition and the flexibility, etc.

	BPEL4WS	**OWL-S**
Domain	Industry driven	Academy driven
Syntactic V.S. semantic	Syntactic description, lacks semantics	Rich in semantics
Interface	Inputs and outputs of the service (described in WSDL)	Input, output, preconditions, and effects (IOPEs)
Flexibility	Has good control over workflow at design time. Not too flexible at runtime	Much more flexible at runtime
Deterministic	Set of choices is pre-determined	Choices are based on goals
Composition	Performed in runtime engine	Planning

Table 1. Comparisons between BPEL4WS and OWL-S

- **WSMO (Web Service Modelling Ontology)**
  WSMO [WSMO] service ontology includes definitions for goals, mediators and web services. A web service consists of a capability and an interface. The underlying representation language for WSMO is F-logic. The rationale for the choice of F-logic is that it is a full first order logic language that provides second order syntax while staying in the first order logic semantics, and has a minimal model semantics. The main characterizing feature of the WSMO architecture is that the goal, web service and ontology components are linked by four types of mediators as follows:
  • OO mediators link ontologies to ontologies,
  • WW mediators link web services to web services,

- WG mediators link web services to goals, and finally,
- GG mediators link goals to goals.

2.3.2 Web Service Domain Ontology

Besides the ontology specially designed to describe Web service, there is another class of ontology that is important and commonly used in describing Web service. It is called Web Service Domain Ontology or simply Domain Ontology**.** It is primarily used to model externally defined knowledge. For example, domain knowledge is used to define the type of functionality the service offers as well as the types of its parameters.

Figure 7 depicts an example of domain ontology used to describe the Web service for finding a medical care supplier. It specifies a Data Structure hierarchy and a Functionality hierarchy. The Functionality hierarchy contains a classification of service capabilities. Two generic classes of service capabilities are shown here, one for finding a medical supplier and the other for calculating distances between two locations. Each of these generic categories has more specialized capabilities either by restricting the type of the output parameters (e.g., find Medicare providers) or the input parameters (e.g., ZipCode, City, SupplyType).

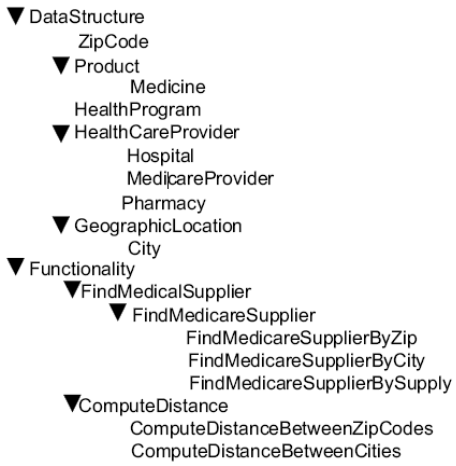

Figure 7: An example of domain ontology

3. Web Service Composition

Web services are well defined and re-usable software components that perform specific, encapsulated tasks via standardized Web-oriented mechanisms. Simple Web services can be assembled together to perform a more complex Web service. This process is called service composition. The result of service composition is the generation of composite services. Service composition can be either performed by compositing primitive or composite services. This allows the definition of increasingly complex applications progressively aggregating components at higher level of abstractions [Dustdar2005].

Service composition is difficult. Its complexity, in general, comes from the following aspects:

- First, the number of services available over the Web increases dramatically during the recent years, and one can expect to have a huge Web service repository to be searched;
- Second, Web services can be created and updated on the fly, thus the composition system needs to detect the updating at runtime and the decision should be made based on the up to date information;
- Third, Web services can be developed by different organizations, which use different concept models to describe the services, however, there does not exist a unique language to define and evaluate the Web services in an identical means.

There are two major classes of techniques for Web service composition. The first class of methods are coming from workflow generation using business process language such as BPEL4WS. The second class of methods try to achieve automatic Web service composition by adding semantic annotations for Web service. AI planning techniques that transform the problem of service composition to a planning problem have been utilized to compose Web services to achieve certain tasks.

3.1 Workflow Composition

In many ways, a composite service is similar to a workflow. The definition of a composite service includes a set of atomic services together with the control and data flow among the services. Similarly, a workflow specifies the flow of work items (e.g. business processes). It is defined as the assembly of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules. Therefore, the current achievements on flexible workflow, automatic process adaption and cross-enterprise integration provide the means for Web services composition as well.

To use workflow technique for service composition, the functionality of a Web service needs to be described with a functional annotation. The industry views Web services as abstract, standardized interfaces to business processes. The specification of a Web service is expressed in WSDL, which specifies only the syntax of messages that enter or leave a computer program. BPEL4WS is used to compose Web service based upon their syntactic functional descriptions in WSDL.

BPEL4WS

BPEL4WS (Business Process Execution Language for Web Service) has become one of the most important technologies of SOA (service-oriented architecture) and enables easy and flexible composition of services into business processes. BPEL4WS is a XML-based workflow definition language that allows businesses to describe enterprise business processes that are connected via Web services. BPEL4WS binds Web services into a single business solution, facilitating their orchestration both within and between enterprises. A business process using BPEL4WS can compose multiple Web services, effectively creating a completely new business application. This grammar can be interpreted and executed by a BPEL orchestration engine.

A business process, as seen by BPEL4WS, is a collection of coordinated service invocations and related activities that produce a result. A BPEL4WS process consists of steps. Each step is called an *activity*. BPEL4WS supports both primitive and structure activities. Primitive activities represent basic constructs and are used for basic tasks, such as those listed below:

- Invoking Web services, using <invoke>
- Waiting for the request, using <receive>
- Manipulating data variables, using <assign>
- Indicating faults and exceptions, using <throw>

We can then combine these activities into more complex processes that specify the steps of a business process. To combine primitive activities, BPEL4WS supports several structure activities. The most important are:

- **Sequence** (<sequence>): Define a set of activities that will be invoked in an ordered sequence;
- **Flow** (<flow>): Define a set of activities that will be invoked in parallel;
- **Case-switch construc**t (<switch>) for implementing branches ;
- **While** (<while>): define loops, etc.

In addition, BPEL4WS provides fault and compensation handlers, event handlers, and correlation sets. It provides means to express complex parallel flows. It also makes it relatively easy to call asynchronous operations and wait for callbacks.

BPEL processes require a runtime environment—a BPEL server, which gives us good control over their execution. Typically, BPEL servers provide control over process instances that are executing and those that have finished. Some of the most popular BPEL servers that are based on Java EE (Sun's new name for J2EE) include Oracle BPEL Process Manager, IBM WebSphere Business Integration Server Foundation, BEA WebLogic Integration, and AquaLogic. There are also at least four open source BPEL servers available: ActiveBPEL Engine, FiveSight PXE, bexee, and Apache Agila.

In BPEL4WS, the messages are simple syntactic descriptions (as an associated XML schema) without any semantics. Such composition processes require manual implementation by the developers. Therefore, the process is hard, time consuming and often error prone.

3.2 **AI Planning for Web Service Composition**

For an automatic web service composition, the semantic-web community draws on AI planning, which for over three decades, has investigated the problem of how to synthesize complex actions, given an initial state, an explicit goal representation and a set of possible state transitions. AI planning considers services as actions and the problem of service composition as a planning problem. It aims to select suitable actions and ordering them in an appropriate sequence so as to achieve some goal.

In general, a classical AI planning problem can be described as a tuple ‹S, S_0, G, A, T›, where

• S is the set of all possible states of the world;
• S_0 denotes the initial state of the world;
• G denotes the goal state of the world the planning system attempts to reach;
• A is the set of actions the planner can perform in attempt to reach a desire goal, and
• The translation relation T=S xAx S defines the precondition and effects for the execution of each action.

For Web services, S_0 and G represent the initial state and the goal state respectively, specified by the service requestors. A is a set of available services and T denotes the current states of each service.

According to Ghallab [Ghallab at.al 2004], AI planning techniques can be categorized into the following classes and sub-classes, as shown in Table 2. The underscored methods in the tables have been applied to compose Web services with varying extent of success.

Classical Planning	• State-Space Planning • Plan-Space Planning
Neoclassical Planning	• Planning-Graph Techniques • Propositional Satisfiability Techniques • Constraint Satisfaction Techniques
Heuristics and Control Strategies	**• Domain-Independent Heuristics** • Control Rules in Planning **• Hierarchical Task Network Planning** **• Situation Calculus** • Dynamic Logic
Planning with Time and Resources (Extensions to (neo-)classical planning)	• Temporal Planning • Planning with Resources
Planning under Uncertainty	• **Planning Based on Markov Decision Processes** **• Planning Based on Model Checking** • Uncertainty with Neoclassical Techniques
Other Approaches to Planning	• Case-based Planning • Multi-agent planning • Plan merging and rewriting • Abstraction Hierarchies • Domain Analysis

Table 2: Categorization of AI planning techniques

The underscored AI planning methods have been applied to compose web services. Amongst them, HTN planning has been shown to be well suited for Web service composition.

In the following, we will discuss two major AI planning methods used for service composition, namely Golog and HTN. We also elaborate on the representative technique in HTN called SHOP2.

3.2.1 Golog Method

In [McIlraith & Son, 2002], a method is presented to compose Web services by applying logical inferencing techniques on pre-defined plan templates. The service are annotated in RDF/OWL-S and then manually translated into Golog. Given a goal description, the logic programming language of Golog [Levesque et al., 1997] is used to instantiate the appropriate plan for composing the Web services. Golog is based on the situation calculus and it supports specification and execution of complex actions in dynamical systems.

Figure 8 shows the generalized plan template taken from [McIl-raith & Son, 2002] that is input to their Golog reasoner. The plan specifies that in order to make a travel booking, i.e., to achieve the goal travel(D1, D2, 0, D), either a return air and subsequent car reservation have to be made between the origin and destination, or a direct car reservation has to be made. This will be followed by making hotel reservations, sending an email about the overall reservation to the user and finally updating the expense claim forms. The Golog reasoner, given the plan templates, evaluates non-deterministic service choices and executes the plan. Golog programs are user-provided plan templates which are customized (bound at runtime) to goal instances. The system uses translator to transform semantic annotations into Golog representations and vice versa.

```
proc(travel(D1, D2, 0, D)
⊏
    ⊏ bookRAirticket(O, D, D1, D2),
      bookCar(D, D, D1, D2)
    ⊐|
    bookCar(O, O, D1, D2),
    bookHotel(D, D1, D2),
    sendEmail,
    updateExpenseClaim
⊐).
```

Figure 8. Travel reservation procedure using Golog. O, D, D1 and D2 stand for Origin, Destination, Departure time and Return time.

3.2.2 HTN Planning

In artificial intelligence, the hierarchical task network, or HTN, is an approach for automated planning. The objective of an HTN planner is to produce a sequence of actions that perform some activity or task. The description of a planning domain includes a set of operators (for each primitive action) similar to those of classical planning, and also a set of methods, each of which is a prescription for how to decompose a task into subtasks (smaller tasks).

HTN Planning is carried out by using the pre-defined methods to decompose tasks *recursively* into smaller and smaller subtasks, until the planner reaches primitive tasks that

can be performed directly using the planning operators. For each non-primitive task, the planner chooses an applicable method, instantiates it to decompose the task into subtasks, and then chooses and instantiates methods to decompose the subtasks even further. If the plan later turns out to be infeasible, the planning system will need to backtrack and try other methods.

Figures 9 shows several methods for performing transporting a package?p, transporting two packages ?p and ?q, dispatching a truck ?t, and returning the truck. In Figure 10, these methods are used to perform the task of transporting two packages ?p and ?q using HTN.

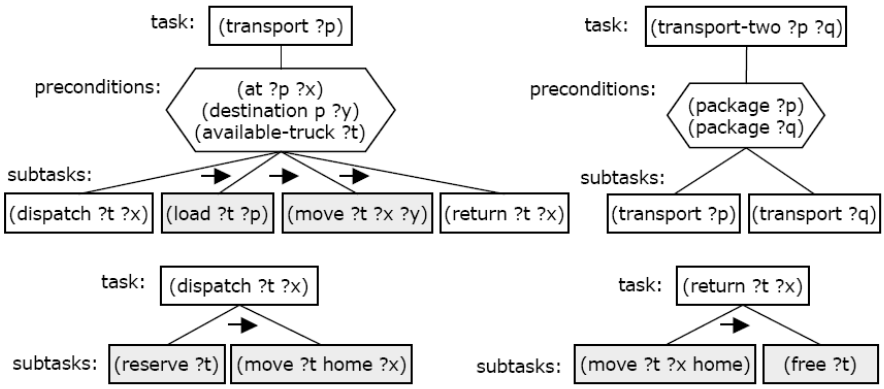

Figure 9. Methods for transporting a package?p, transporting two packages ?p and ?q, dispatching a truck ?t, and returning the truck. Arrows are ordering constraints. The shaded subtasks are primitive tasks that are accomplished by the following planning operators: (load ?t ?p) loads ?p onto ?t; (move ?t ?x ?y) moves ?t from ?x to ?y; (reserve ?t) deletes (available-truck ?t) to signal that the truck is in use; (free ?t) adds (available-truck ?t) to signal that the truck is no longer in use.

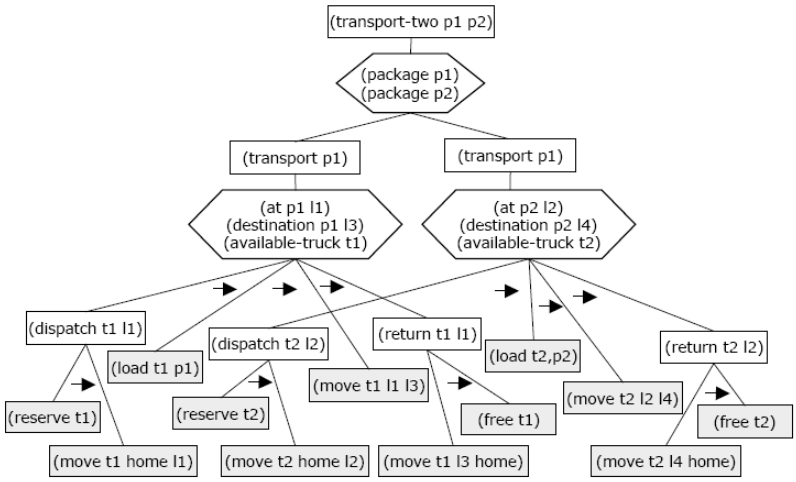

Figure 10. A plan for accomplishing (transport-two p1 p2) from the following initial state: {(package p1), (at p1 l1), (destination p1 l3), (available-truck t1), (at t1 home), (package p2), (at p2 l2), (destination p2 l4), (available-truck t2), (at t2 home)}.

- **Representative method for HTN: SHOP and SHOP2**

SHOP stands for Simple Hierarchical Ordered Planner. SHOP2 [Nau 2003] [Sirin 2004] is the latest version of SHOP. SHOP2 is based on SHOP [Nau et al., 1999], a previous domain-independent ordered task decomposition planner that requires the subtasks of each method, and also the initial set of tasks for the planning problem, to be totally ordered rather than partially ordered. In SHOP, subtasks of different tasks cannot be interleaved. SHOP2 extends SHOP by allowing the subtasks of each method to be partially ordered. This makes SHOP2 to create plans more efficiently than SHOP, using domain descriptions simpler than those needed by SHOP. Both SHOP and SHOP2 are available as open-source software at hhttp://www.cs.umd.edu/projects/shopi.

A. Domain Descriptions in SHOP2

A domain description is a description of a planning domain, consisting of a set of *methods*, *operators*, and *axioms*.

- **Tasks**

A task represents an activity to perform. Syntactically, a task consists of a task symbol followed by a list of arguments. A task may be either primitive or compound. A primitive task is one that is supposed to be accomplished by a planning operator: the task symbol is the name of the planning operator to use, and the task's arguments are the parameters for the operator. A compound task is one that needs to be decomposed into smaller tasks using a method; any method whose head unifies with the task symbol and its arguments may potentially be applicable for decomposing the task.

- **Operators**

Each operator indicates how a primitive task can be performed. Each operator o has a head head(o) consisting of the operator's name and a list of parameters, a precondition expression pre(o) indicating what should be true in the current state in order for the operator to be applicable, and a delete list del(o) and add list add(o) giving the operator's negative and positive effects.

- **Methods**

Each method indicates how to decompose a compound task into a partially ordered set of subtasks, each of which can be compound or primitive. The simplest version of a method has three parts: the task for which the method is to be used, the precondition that the current state must satisfy in order for the method to be applicable, and the subtasks that need to be accomplished in order to accomplish that task.

- **Axioms**

The precondition of each method or operator may include conjunctions, disjunctions, negations, universal and existential quantifiers, implications, numerical computations, and external function calls. Furthermore, axioms can be used to infer preconditions

that are not explicitly asserted in the current state. The axioms are generalized versions of Horn clauses, written in a Lisp-like syntax: for example, (:- head tail) indicates that head is true if tail is true. The tail of the clause may contain anything that may appear in the precondition of an operator or method.

```
(:method
  ; head
       (transport-person ?p ?c2)
  ; precondition
       (and
         (at ?p ?c1)
         (aircraft ?a)
         (at ?a ?c3)
         (different ?c1 ?c3))
  ; subtasks
       (:ordered
         (move-aircraft ?a ?c1)
         (board ?p ?a ?c1)
         (move-aircraft ?a ?c2)
         (debark ?p ?a ?c2)))
```

Figure 11. An example of method in SHOP2

```
(:-
  ; head
    (enough-fuel ?plane ?current-position ?destination ?speed)
  ; tail
    (and (distance ?current-position ?destination ?dist)
         (fuel ?plane ?fuel-level)
         (fuel-burn ?speed ?rate)
         (eval (>= ?fuel-level (* ?rate ?dist)))))
```

Figure 12. An example of Axiom in SHOP2

B. Algorithm of SHOP2

Figure 13 shows a simplified version of the SHOP2 planning procedure. The arguments include the initial state *s*, a partially ordered set of tasks *T*, and a domain description *D*.

SHOP2 plans for tasks in the same order that they will be executed. In order to do this, it chooses a task *t* from *T* that has no predecessors; *t* is the first task that SHOP2 will work on. At this point, there are two cases.

1. The first case is if *t* is primitive, i.e., if *t* can be accomplished directly using an action (i.e., an instance of a planning operator). In this case, SHOP2 finds an action *a* that matches *t* and whose preconditions are satisfied in *s*, and applies *a* to *s* (if no such action exists, then this branch of the search space fails).

2. The second case is where *t* is compound, i.e., a method needs to be applied to *t* to

decompose it into subtasks. In this case, SHOP2 chooses a method instance *m* that will decompose t into subtasks (if no such method instance exists, then this branch of the search space fails). The last three lines of the loop ensure that this will happen, by telling SHOP2 that if the current method *m* has any subtasks, SHOP2 should generate one of those subtasks before generating any other subtasks.

```
procedure SHOP2(s, T, D)
  P = the empty plan
  T_0 ← {t ∈ T : no other task in T is constrained to precede t}
  loop
    if T = ∅ then return P
    nondeterministically choose any t ∈ T_0
    if t is a primitive task then
      A ← {(a, θ) : a is a ground instance of an operator in D, θ is a substi-
            tution that unifies {head(a), t}, and s satisfies a's preconditions}
      if A = ∅ then return failure
      nondeterministically choose a pair (a, θ) ∈ A
      modify s by deleting del(a) and adding add(a)
      append a to P
      modify T by removing t and applying θ
      T_0 ← {t ∈ T : no task in T is constrained to precede t}
    else
      M ← {(m, θ) : m is an instance of a method in D, θ unifies {head(m), t},
            pre(m) is true in s, and m and θ are as general as possible}
      if M = ∅ then return failure
      nondeterministically choose a pair (m, θ) ∈ M
      modify T by removing t, adding sub(m), constraining each task
        in sub(m) to precede the tasks that t preceded, and applying θ
      if sub(m) ≠ ∅ then
        T_0 ← {t ∈ sub(m) : no task in T is constrained to precede t}
      else T_0 ← {t ∈ T : no task in T is constrained to precede t}
  repeat
end SHOP2
```

Figure 13. Algorithm of SHOP2

References

[Borst, 1997] Borst, W. (1997). *Construction of Engineering Ontologies*. PhD thesis, University of Tweente, Enschede, NL–Centre for Telematica and Information Technology.

[Studer et al., 1998] Studer, R., Benjamins, V., and Fensel, D. (1998). Knowledge Engineering: Principles and Methods. *Data and Knowledge Engineering*, 25(1-2):161 – 197

[Horrocks] I. Horrocks, D. Fensel, J. Broekstra, M. Crubezy, S. Decker, M. Erdmann,W. Grosso, C. Goble, F. Van Harmelen, M. Klein, M. Musen, S. Staab, and R. Studer. The ontology interchange language oil: The grease between ontologies. http://www.cs.vu.nl/˜dieter/oil.

[Woods1992] W. A.Woods and J. G. Schmolze. The KL-ONE Family. *Computers Math. Applic.*, 23(2-5):133.177, 1992.

[Bechhofer1999] S. Bechhofer and C.A. Goble. Delivering Terminological Services. *AI*IA Notizie, Periodico dell'Associazione Italiana per l'intelligenza Articiale*, 12(1), March 1999.

[Farquhar1997] A. Farquhar, R. Fikes, and J.P. Rice. The ontolingua server: A tool for collaborative ontology construction. *Journal of Human-Computer Studies*, 46:707.728, 1997.

[Duce1988] D.A. Duce and G.A. Ringland. *Approaches to Knowledge Representation: An Introduction*. Knowledge-Based and Expert Systems Series. John Wiley, Chichester, 1988.

[Noy 2000] N.F. Noy, R.W. Fergerson, M.A. Musen, The knowledge model of protege-2000: combining interoperability and flexibility, in: 12th International Conference in Knowledge Engineering and Knowledge Management (EKAW'00), Lecture Notes in Artificial Intelligence, vol. 1937, Springer, Berlin, 2000, pp. 17–32.

[Noy_1 2000] N.F. Noy, M.A. Musen, PROMPT: Algorithm and Tool for Automated Ontology Merging and Alignment, in: 17th National Conference on Artificial Intelligence (AAAI'00), Austin, 2000.

[Arpırez2001] J.C. Arpırez, O. Corcho, M. Fernandez-Lopez, A. Gomez-Perez, WebODE: a scalable ontological engineering workbench, in: First International Conference on Knowledge Capture (KCAP'01), ACM Press, Victoria, 2001, pp. 6–13.

[Corcho2002] O. Corcho, M. Fernandez-Lopez, A. Gomez-Perez, O. Vicente, WebODE: an integrated workbench for ontology representation, reasoning and exchange, in: A. Gomez-Perez, V.R. Benjamins (Eds.), 13th International Conference on Knowledge Engineering and Knowledge Management (EKAW'02), Lecture Notes in Artificial Intelligence, vol. 2473, Springer, Berlin, 2002, pp. 138–153.

[Decker1999] S. Decker, M. Erdmann, D. Fensel, R. Studer, Ontobroker: Ontology based access to distributed and semistructured information, in: Semantic Issues in Multimedia Systems (DS8), Kluwer Academic Publisher, Boston, 1999, pp. 351–369.

[Sure2002] Y. Sure, M. Erdmann, J. Angele, S. Staab, R. Studer, D. Wenke, OntoEdit: collaborative ontology engineering for the semantic web, in: First International Semantic Web Conference (ISWC'02), Lecture Notes in Computer Science, vol. 2342, Springer, Berlin, 2002, pp. 221–235.

[Nau 1999] Nau, D., Cao, Y., & Mu˜noz-Avila, H. (1999). SHOP: Simple Hierarchical Ordered Planner. In IJCAI-99, pp. 968–973.

[Nau 2003] Dana S. Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, Fusun Yaman: SHOP2: An HTN Planning System. J. Artif. Intell. Res. (JAIR) 20: 379-404 (2003).

[Sirin 2004] Evren Sirin, Bijan Parsia, Dan Wu, James A. Hendler, Dana S. Nau: HTN planning for Web Service composition using SHOP2. J. Web Sem. 1(4): 377-396 (2004)

[Levesque et al., 1997] Levesque, H. J.; Reiter, R.; Lesperance, Y.; Lin, F.; and Scherl, R. B. 1997. GOLOG: A logic programming language for dynamic domains. Journal of Logic Programming 31(1-3):59-83.

[McIlraith & Son, 2002] McIlraith, S., and Son, T. C. 2002. Adapting golog for composition of semantic web services. In Proc. KRR, 482-493.

[Ghallab at.al 2004]Malik Ghallab, Dana Nau, and Paolo Traverso, Automated Planning: Theory and Practice, Amsterdam Morgan Kaufmann Publisher, 2004.

[WSMO] http://www.wsmo.org/

[OWL-SProfile] http://www.daml.org/services/owl-s/1.0/Profile.owl

[OWL-SProcess] http://www.daml.org/services/owl-s/1.0/Process.owl

[OWL-SGrouding] http://www.daml.org/services/owl-s/1.0/Grounding.owl

[OWL-SService] http://www.daml.org/services/owl-s/1.0/Service.owl

[Martin] Martin, D., Burstein, M., Denker, G., Hobbs, J., Kagal, L., Lassila, O., McDermott, D., McIlraith, S., Paolucci, M., Parsia, B., Payne, T., Sabou, M., Sirin, E., Solanki, M., Srinivasan, N., and Sycara, K. (2003). OWL-S 1.0 white paper. http://www.daml.org/services/owl-s/1.0/.

[Motta2003] Motta, E., Domingue, J., Cabral, L., and Gaspari, M. (2003). IRSII: A Framework and Infrastructure for Semantic Web Services. In (Fensel et al., 2003), pages 306 – 318.

[RiboWeb]http://smi-web.stanford.edu/projects/helix/riboweb.html

[Dustdar2005] S. Dustdar and W. Schreiner. A survey of web services composition. Internationall Journal of Grid Services, Vol 1, No. 1, 2005.

[Curbera & others, 2002] Curbera, F., et al. 2002. Business process execution lan- guage for web services. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.

[Arkin & others, 2002] Arkin, A., et al. 2002. Web services choreography interface WSCI. http://www.w3.org/TR/wsci/.

[Berners-Lee, Hendler, & Lassila, 2001] Berners-Lee, T.; Hendler, J.; and Lassila, O. 2001. The semantic web. Scienti_c American, May issue.

[Christensen & others, 2001] Christensen, E.,et al. 2001. The web services description language WSDL. http://www-4.ibm.com/software/solutions/webservices/resources.html.

[McBride, 2004] McBride, B. (2004). The Resource Description Framework (RDF) and its Vocabulary Description Language RDFS. In (Staab and Studer, 2004), pages 51 – 66.

[Horrocks et al., 2003] Horrocks, I., Patel-Schneider, P., and van Harmelen, F. (2003). From SHIQ and RDF to OWL: The Making of a Web Ontology Language. *Journal of Web Semantics*, 1(1):7 – 26.

[Klyne2004] Graham Klyne and Jeremy J. Carroll, Editors. "Resource Description Framework (RDF): Concepts and Abstract Syntax". W3C Recommendation 10 February 2004. http://www.w3.org/TR/rdf-concepts/

[Bray 2000] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler, Editors. "Extensible Markup Language (XML) 1.0 (Second Edition)". W3C Recommendation 6 October 2000. http://www.w3.org/TR/REC-xml/

[Lassila2002] Lassila, O. (2002). Serendipitous Interoperability. In (Hyvonen, 2002).

[UDDI] The UDDI Technical White Paper.
http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf

[OASIS] http://www.oasis-open.org/home/index.php

[TaO] http://img.cs.man.ac.uk/tambis

[Brazma2001] Brazma A, Hingamp P, Quackenbush J, *et al*. 2001. Minimum information about a microarray experiment — MIAME — towards standards for microarray data. *Nature Genet* 29: 365–371.

[MGED] Microarray Gene Expression Data Society: http://www.mged.org

[OMG] Object Management Group: http://www.omg.com/

[Spellman2002] Spellman P, Miller M, Stewart J, *et al*. 2002. Design and implementation of microarray gene expression markup language (MAGE-ML). *Genome Biol* 3: RESEARCH0046.

[Snobase] http://www.alphaworks.ibm.com/tech/snobase/

[Uschold1996]M. Uschold and M. Gruninger. Ontologies: Principles, Methods and Applications. *Knowledge Engineering Review*, 11(2), June 1996.

[Gomez-Perez1994]A. Gomez-Perez. Some Ideas and Examples to Evaluate Ontologies. Technical Report Technical Report KSL-94-65, Knowledge Systems Laboratory , Stanford, 1994.

[GO] http://genome-www.stanford.edu/GO/

[MetaCyc] www.metacye.org

[BioPAX] www.biopax.org

[KEGG] www.genome.ad.jp/kegg

[Ecocyc] www. ecocye. Org

[Noy_2] Natalya F. Noy and Deborah L. McGuinness. Ontology Development 101: A Guide to Creating Your First Ontology, Stanford University.