View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DigitalCommons@CalPoly



# On the Road to Intelligent Web Applications

**Dr Hisham Assal**
CAD Research Center, Cal Poly University

**Kym J. Pohl**
CDM Technologies Inc.

**Abstract**

Increasing access to data sources on the Internet offers expanding opportunities for equipping intelligent applications with the content they require whether broad in scope or rich in detail. Although typically originating within the web in a *semi-structured* form, with the use of inference-based translation and analysis mechanisms such content can be transformed into useful information and ultimately into actionable knowledge. Service-Oriented Architecture (SOA) offers a platform for accessing the web as invocable resources and effectively incorporating multiple sources of data and capabilities on the Internet into enterprise applications. Adding inference capabilities to SOA-based applications not only aids in the translation of data into information thus increasing visibility into the sea of content that is the *web*, but also provides a powerful mechanism for performing the domain-centric decision making that is the heart of intelligent applications. The Web Ontology Language (OWL) offers the medium and the tools necessary to represent models of business activities as well as support native inference across related semantic concepts. In this paper the authors present an architecture for combining OWL with a SOA-based paradigm to enhance traditional web applications with powerful inference capabilities. Commensurate with a service-oriented theme, specific techniques are presented for representing the translation activity itself as a service. The paper concludes with a discussion of two distinct types of inference: one internal to the OWL model and the other externalized into intelligent agents that operate across OWL-based concepts.

**Keywords:** inference, SOA, OWL, intelligent analysis, web application, semantic web

# Introduction

Web applications strive to take advantage of the sea of content available on the Internet. With the migration of Service Oriented Architecture (SOA) into the modern day Web, there is a growing trend towards exposing such content as web services (McKendrick 2009). Although certainly abundant in the financial, media, and search domains, standardized protocols in conjunction with painless development platforms this proliferation of web services is far-reaching into even the most informal Internet communities. In order to effectively function within this service-oriented playing field, web applications must be architected in a manner that is compatible with these service-based environments and related SOA principles (Erl 2008).

However, compatibility with a service-oriented environment is unfortunately not sufficient to effectively take advantage of the vast amounts of content available on the web. Once obtained, the limited representation of this sought-after content offers yet another hurdle. The painful reality of today's web is that the vast majority of this content is at best found in a semi-structured form and is usually no more than sections of free text. For web applications intending to apply some reasonable degree of analysis on this content, practicality in today's data-centric Internet requires these applications to infer meaning from this otherwise context-deprived data. To support such an activity, the service-oriented fabric within which these web applications operate should be equipped with a facility capable of supporting various forms of inference. The Web Ontology Language (OWL) offers the provisions for developing semantically-enriched models where domain concepts can be represented in not only structure but also logic allowing platform-level reasoners (Bock et. al. 2008) to perform the inference activities necessary to make sense out of the sea of data that is today's Internet.

The utility of incorporating an inference capability into the platform within which web applications operate goes beyond the transformation of data into information. Whether transformed from data or originating within the context-rich Web that is the promise of the Semantic Web (Berbers-Lee et. al. 2001), once web-based content is available as *information* an entirely new set of decision-support possibilities becomes apparent. Within this environment, web applications will engage in considerably more reasoning activities than the web applications of the past. Equipping the framework within which web applications operate with the constructs and facilities that directly support such inference activities will be imperative.

Following is a discussion presenting several powerful features of OWL that can be leveraged by web applications to perform the inference necessary to transform data into information, as well as capitalize on this information to perform sophisticated analysis. The reader is then presented with a hybrid architecture that successfully integrates an OWL execution platform with a service-oriented architecture managing the bridge between these two distinct paradigms. The paper concludes with a discussion of key distinctions between a *service* and an *intelligent agent*

# OWL: Web Ontology Language

The Web Ontology Language, commonly referred to simply as OWL, is a semantic markup language. The primary purpose of OWL is to facilitate the publishing and sharing of ontologies across the World Wide Web (WWW). OWL is intended to be used by software applications that need to process web-based content in a meaningful manner. In other words, OWL-based content is designed to be machine-interpretable.

A typical OWL environment consists of several key components, some to be employed at development-time and others that manage runtime activities. Together, these components form a cohesive platform for the development and execution of semantic content.

## OWL Modeler

The OWL paradigm supports a number of powerful modeling concepts including dynamic, multiple classification as well as unconstrained attribute composition. As such, developing a model that takes full advantage of such features requires a development environment equipped with native and intuitive support for key modeling constructs. Further, such modeling environments should seamlessly integrate with model validation, presentation, reasoning, and code generation capabilities. There are a variety of such development tools available in off-the-shelf form including Protégé.

## OWL Reasoner

Perhaps the most important component of any OWL environment is the *reasoner*. As the name implies, the main function of this component is to essentially reason about a given OWL model and its associated content. More specifically, an OWL *reasoner* processes class definitions, individuals, and rules in an effort to accomplish two primary objectives:

1. To identify any logical inconsistencies existing within the model definition and its use. Some of these inconsistencies may take the form of uninstantiable classes, conflicting definitions, and so on.
2. To identify any additional knowledge that can be automatically inferred based on the model definition and associated content. This additional knowledge can include subsumption and association relationships or the qualification of an individual for membership to under classification(s). For example, based on class definitions, one class may meet all of the criteria to be considered a subclass of another. Likewise, based upon the characteristics of a particular individual, that individual may also meet the requirements to be a member of one or more additional classifications. It should be noted that most reasoners available off-the-shelf (OTS) focus on inferring *additional* knowledge, and not necessarily managing the validity, or truth, of existing knowledge. In other words, most OTS reasoners make little attempt to retract inferred knowledge once it is no longer valid. Managing the truth of such classifications is vital in maintaining an accurate account of inferred knowledge. As such, the inability of most OTS reasoners to perform this maintenance is a serious limitation to their practical use and results in such maintenance being the responsibility of the developer.

## OWL Query Engine

The ability to interrogate or ask questions of an executing OWL model is a core requirement of any knowledge-based system. In fact, this is typically the primary means by which inferencing is performed within such environments. Although certainly not a requirement, this facility is often integrated into the OWL *reasoner* itself. Such intermingling of these two capabilities makes sense since processing queries within an OWL-based paradigm often requires degrees of inferencing.

Apart from a powerful query engine, an appropriate query language must be selected and consequently supported. Such language should be powerful enough to support representing the

semantic-level questions that are often posed within an OWL environment. Such questions often go beyond the classical SQL-level queries and take forms such as "is *Jennifer* a cousin of *Luke*?" or "what's an appropriate diagnosis for these symptoms?". Both of these examples would typically require the use of a *reasoner* in order to formulate appropriate answers.

## Key Concepts Promoted by OWL

OWL supports several very powerful concepts. Although certainly not unique to the OWL paradigm, these concepts are the fundamental enablers of OWL's support for semantic-oriented representation (i.e., models). Following is a discussion of each of these core concepts.

***Multiple Classification:*** As the name implies, multiple classification is the ability for an entity to be classified as one or more types simultaneously. This is a very powerful capability and has significant implications on the manner in which representational models are developed. Unlike traditional, more rigid modeling paradigms where inheritance must be employed in order to extend abstract classifications, OWL modelers enjoy a very flexible environment without concern for relating classifications in order to support a single object exhibiting features defined across multiple classifications. To manage exactly which classifications are appropriate is typically the responsibility of the OWL reasoner. Comparing features exhibited by objects against requirements for class membership, the OWL reasoner can determine which classifications a particular object currently qualifies for.

***Dynamic Classification:*** Dynamic classification is the ability of the classification of an object to change over time. Different than re-instantiating an entity under a new classification, the identity's referential integrity is preserved as its classification(s) change throughout time. This capability goes hand-in-hand with multiple classification and together these concepts create a very dynamic environment where objects can effectively mutate throughout their lifecycle. Like management of multiple classification, determining exactly what classification(s) an OWL object qualifies for at any point in time is typically the responsibility of the OWL reasoner.

***Open World Assumption (OWA):*** Traditional database systems operate under a set of assumptions to enable the query engine to return meaningful response. These assumptions include: the closed world assumption; the unique name assumption; and, the domain closure assumption. The closed world assumption states that if a statement cannot be proven true, given the information in the database, then it must be false. The unique name assumption states that two distinct constants designate two different objects in the universe. The domain closure assumption states that there are no other objects in the universe than those designated by constants of the database.

These assumptions were reasonable in a world where a database represented all of the information available about a given domain and no external information sources were needed to perform the functions of any database application. However, with the Internet becoming a major source of information, many applications are based on access to external information from sources that may be unknown at the design stage of the application. This requires a different kind of knowledge representation, capable of dealing with the openness of the Internet. The open world assumption was adopted to allow for

the relaxation of the constraints of the closed world assumption. Along with the open world assumption, the other two assumptions were also relaxed, namely, the unique name assumption and the domain closure assumption.

The open world assumption states that there can be true statements that are not contained in the current representation. The unique name assumption is dropped to allow two objects to have the same name without being considered the same object. This means that two objects are considered the same, only if there is a statement that they are. The domain closure assumption was relaxed and converted to an open domain assumption, which states that there can be other objects in the universe than those in the current representation, unless explicitly stated otherwise.

Under an open world assumption, everything is possible unless asserted otherwise. This is in stark contrast to traditional decision-support paradigms where the volume and extent of considerations is limited to what is explicitly asserted to be true about the *world* at any given time. Although operating under an open world assumption has implications on model development, it is primarily model usage and interpretation that are affected. For example, unless stated otherwise it is certainly conceivable that two otherwise distinct objects actually represent the same entity. This simple yet powerful implication can affect whether a reasoner determines an inconsistency regarding two individuals (i.e., objects) being assigned to the same end of a functional (i.e., *a multiplicity of one*) property (i.e., relationship) or inferring that these two individuals are actually the same.

***Unconstrained Composition of Characteristics:*** Being able to assign characteristics to individuals in a manner unbounded by the blueprint of the individual's current classification(s) is a primary trigger for dynamic classification. In environments supporting this flexibility users are free to assign to and remove characteristics from objects regardless of the object's current definition or type(s). The extent of available characteristics is bound only by the range of types defined within the model together with the avoidance of any inconsistencies as prescribed by model logic. As an individual's characteristics are changed, so may the set of classification memberships the individual qualifies for. Determining exactly what changes in classification are appropriate is the responsibility of the reasoner. For example, consider that having certain characteristics, *Rusty* currently meets the qualifications to be a Person. Suppose now that Rusty is also asserted to have a tail, a feature that is not part of the class definition of *Person*. The reasoner may now determine that *Rusty* should no longer be considered a *Person*, but rather a *Dog*. This kind of type-relaxation can be a powerful means for automatically (i.e., at the framework-level) adjusting to changing conditions and is one of the most powerful features of an OWL environment.

Perhaps the most exciting aspect of environments supporting the concepts described above is the ability to off-load significant amounts of semantic processing to framework-level components (i.e., the reasoner, etc.). Activities such as managing appropriate classification which, if supported at all, were traditionally the responsibility of application-level components can now be transparently managed by the framework itself. Not only does this result in significantly less work by application developers but by internalizing such activities can lead to improved performance compared to more externalized approaches.

## Supporting Architectures

It is important to realize that the concepts described above are not necessarily unique to OWL. Rather, OWL is only one environment that supports such concepts. It is certainly possible to implement concepts such as dynamic, multiple classification and unconstrained composition within traditional object-oriented environments as well. There are numerous modeling patterns and techniques that can be employed in support of such capabilities. Figure 1 provides a Unified Modeling Language (UML) model fragment illustrating how some of these concepts can be readily represented within more rigid modeling paradigms. This model fragment has two distinct sides, a knowledge side that essentially represents type information, and an operational side that represents individual entities. The model presented in Figure 1 can be read as follows.

> There exist various types of things (i.e., *ThingType*). These types have varying degrees of compatibility with each other (i.e., *isDisjointWith*) as well as the types of roles (i.e., *RoleType*) that things of these types can potentially play (i.e., *canPlay*). Note that additional knowledge-level constraints can be added to the model fragment in a similar manner representing notions such as symmetry, transitivity, irreflexibility, and so on. The operational side of the model indicates that a specific thing (i.e., *Thing*) can be classified simultaneously under one or more classifications (i.e., *ThingType*). Further, depending on its characteristics a thing can change type(s) dynamically throughout time without jeopardizing its unique identity. In addition, things can play a variety of roles (i.e., *Role*) throughout time. These roles are typed according to their specific *RoleType*. The set of potential roles a thing can play is governed by the thing's current set of classifications.

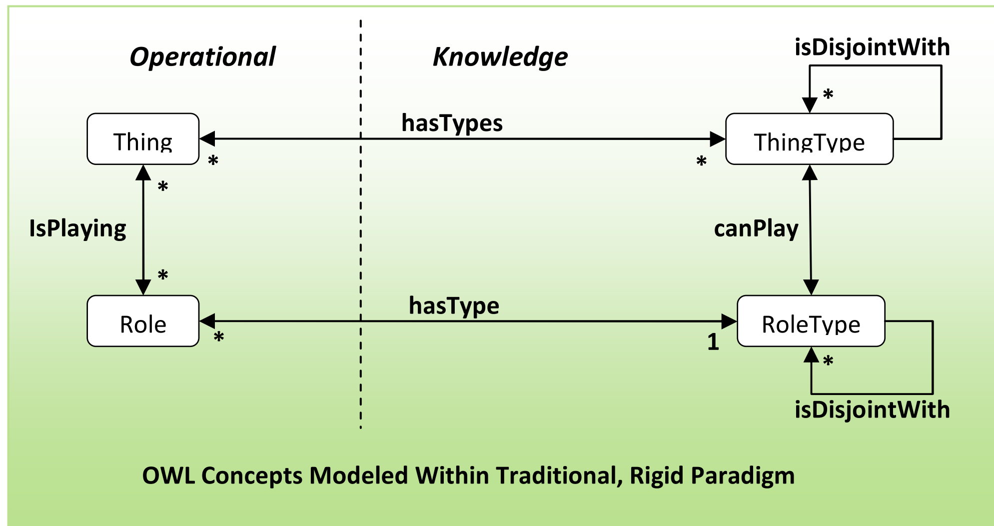


Figure 1: UML pattern supporting dynamic, multiple classification and type compatibility

However, despite the mechanics supportable within more traditional, rigid environments there is a difference between an environment where such support is *possible* and one where it is *native* to the platform. It is important that the mechanics of such support be as transparent and integrated as possible alleviating users from dealing with supporting model elements that are not directly aligned with target concepts. Although by no means unique, OWL's native support for such

concepts should not go unnoticed and can translate into not only significant savings in development costs but can also result in a more elegant implementation.

However, a solely OWL-based solution may not be adequate when all factors are considered. A more hybrid approach may, in fact, offer a more balanced and successful result. The reasons for this are as follows. First, although a promising direction, there is a significant difference between the extensiveness of support and maturity of traditional enterprise environments as compared to that of OWL. Second, due in no small part to this maturity, traditional enterprise frameworks tend to be notably more refined and efficient when compared to current OWL offerings. Further, many organizations have a considerable investment in traditional toolsets and frameworks and are therefore reluctant to completely abandon such capabilities in favor of a pure OWL solution. Accordingly, it can be argued that an effective architecture should take the form of a partnership between an OWL platform and one that comprises more traditional components. The objective of such a solution would be to promote the advantages of both environments while minimizing their limitations. The following section presents a reference architecture, which combines components that natively support OWL with those inherent in a more traditional, rigid environments.

## Hybrid Architecture

The objective of the architecture proposed in this section is to combine the emerging support for OWL with the mature and extensively supported object-oriented enterprise environments currently employed by numerous organizations. The resulting architecture strives to capitalize on the benefits exhibited by each individual paradigm by segregating specific tasks to appropriately-suited mechanisms. Since this architecture exists as a sort of cross between two existing architectures, the resulting combination is referred to as a *hybrid* architecture.

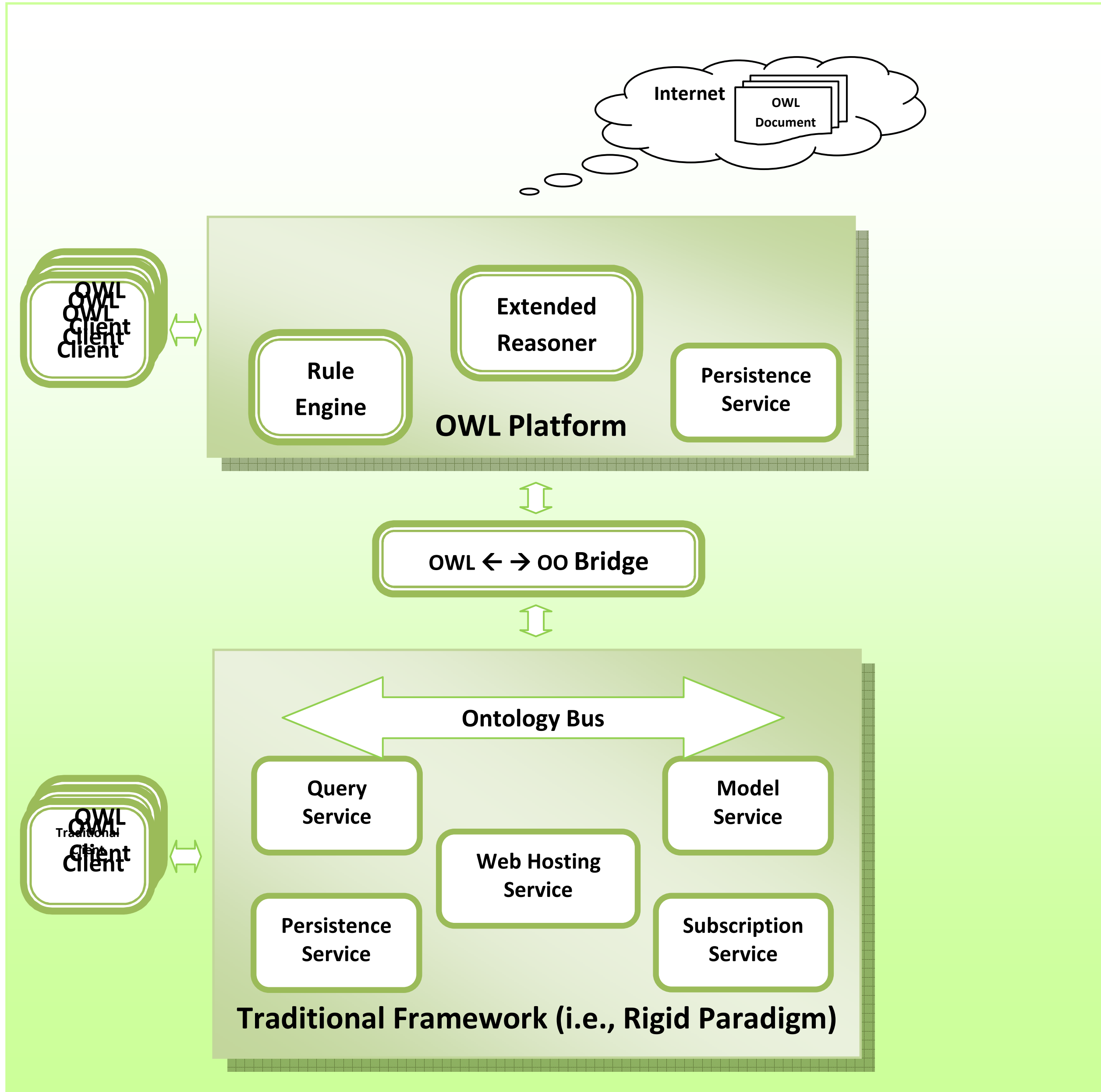


Figure 2: Hybrid architecture integrating more traditional components with those of OWL

### The OWL Side

Considering the native manner in which the OWL platform supports the powerful concepts described earlier, it is imperative that the solution architecture include the fundamental components comprising an OWL architecture. As mentioned earlier, these components include an OWL modeler, reasoner, rule engine, and query processor. Ideally, these components would be taken from off-the-shelf offerings. However, at the time of this writing OWL is still undergoing a significant maturing process. As such, it is likely that some additional development will be necessary to elevate certain off-the-shelf components to the required level of support. It is especially imperative that the employed reasoner not only support truth maintenance but that it

performs its inference on an asynchronous and continual basis. Further, the reasoner must be able to perform this monitoring across potentially large portions of model elements and, considering the frequency with which model content is likely to change in the complex, multivariable environment of decision-support operations, it must do so in as efficient a manner as possible.

### The Traditional Side

The traditional side of the equation is characterized by its scalable framework directly supporting the business processes of a complex, multi-faceted organization. At the heart of the more modern-day variety are one or more well-crafted, object-oriented domain models providing the context upon which enterprise-level, decision-support activities are performed. However, although over the years enterprise application frameworks have certainly proven their worth, they do not traditionally support the level of flexibility and dynamics available in OWL.

This can be seen in the high degree of rigidity typically found with the domain models that they manage. Concepts such as dynamic classification and multiple classification, let alone unconstrained composition, are not natively supported in such paradigms and therefore are not typically engineered into the models that these environments manage. Although some of these concepts are, in fact, supportable within such rigid modeling paradigms, *explicit* representation of such concepts (Figure 1) typically adds significant complexity and overhead to already extensive domain models. Further, such environments offer limited support for the management of such concepts. Such frameworks contain limited facilities for managing the concepts that are core to OWL, including the dynamic reclassification of an entity. While such support can be developed within framework-level components, this would typically introduce notable overhead. As a result, the exposure of elaborate model constructs to users increases the overall complexity and dilutes the otherwise domain-centric nature of the model with extraneous notions (i.e., a role, an entity-property relationship, and so on). Even if such intricate complexities were to be overlooked, the models would need to be re-architected to achieve compatibility with notions such as dynamic classification and unconstrained composition.

Considering the difficulties described above, a more realistic objective would appear to be to focus support on those concepts that directly address core use-cases found in today's decision-support systems. Although not comprehensive, this list would include the ability to support multiple views of an entity personalized to the native vocabulary, structure, and scope of individual users. To avoid the need to re-architect such users for compatibility, it may suffice to limit the scope of such multiple and dynamic classification to exist across users, and not necessarily within the scope of an individual user itself. Easing of this scope allows for users operating within platforms not directly supporting these concepts to still effectively *play* within such arenas.

### The Bridge

As its name indicates, the primary purpose of the Bridge is to form a connective conduit between the two platforms, or paradigms comprising the overall architecture. Functioning much like a basic messaging service, the Bridge fulfills requests to send content from one environment to the other. To accomplish this feat, however, the Bridge must typically perform a level of translation together with occasional orchestration in order to effectively and correctly represent the content within the neighboring paradigm. Although each environment may manage its own set of native model fragments, one of the goals of this approach is to facilitate the modeling of cross-

environment domain concepts within the more powerful modeling paradigm offered by OWL. With such an approach, any model fragment necessary to represent such concepts within the more rigid traditional side of the architecture would be automatically derived from the original OWL-based description. Although these shared domain concepts stem from the original OWL-based incarnation, their composition can understandably differ significantly. Within the OWL environment, such concepts are represented as a natural part of the OWL language. Whereas, in the more rigid modeling paradigm such concepts are supported through employment of specific analysis patterns (Figure 1) and consequently managed through purpose-built extensions to framework-level components (e.g., a Model Server capable of supporting multiple views, or facades).

### Translating Classification

One of the core activities this hybrid architecture must support deals with passing changes in an entity's classification from one side of the architecture to the other. More specifically, the architecture must support a complex set of activities ranging from the initial determination of an entity's classification(s) to the translation and consequential mirroring of such an event within the neighboring world. As discussed earlier, the determination and consequential management of classification is a capability readily supported by any reasoner-equipped OWL platform. As such, management of an entity's classification(s) should clearly be handled by the OWL side of the equation. Therefore, it is the task of the Bridge to translate changes in an entity's classification into the model element counterparts (i.e., facades or views) offered within the traditional environment.

To help convey the key steps involved in this translation process, consider the example scenario of a weather system beginning to impede the use of a frequented section of roadway. Within the OWL environment the reasoner quickly determines that the OWL individual representing the weather system should not only be categorized under its original WeatherSystem classification but should now also qualify as a *TrafficImpediment*, for example. Reacting to this additional classification, the Bridge has the task of reflecting this event within the traditional side of the architecture ready for consumption by traditional enterprise components. As described above, within the more rigid modeling paradigm governing the traditional side of the architecture, additional classifications of an entity are represented as stateful facades, or views, overlaid upon the original entity. In this example, the original entity representing the weather system might be an instance of the *WeatherEvent* class and an impediment-oriented view of such an entity might be represented as a *RoadUsageImpediment* that derives its weather-related properties from the underlying WeatherEvent model fragment. The impediment-related properties would be the stateful part of the façade or view. Once translated into this form, users operating within the more rigid environment that are interested in seeing the weather system in its innate form would interact with it as a WeatherSystem entity. By the same token, users only interested in things that impede traffic would interact with the entity as a RoadUsageImpediment. In either case, through support for this type of OWL-like classification, users would have the ability to see and interact with entities in the form most suitable for their individualized perspectives.

This example illustrates how *hybrid* architecture capitalizes on the dynamic, multiple classification capabilities inherent within OWL in a manner compatible with somewhat more rigid, yet notably more resourced and utilized, traditional platforms. The Bridge component of

the architecture provides a seamless conduit between both worlds whereby events and affects in one environment can be effectively reflected in the other.

## Service-Oriented Architectures and Agent-Based Systems

The semantic web is promoted as an environment in which meaningful exchange of information can take place. Both service-oriented architectures and agent-based systems are considered paradigms that fit the semantic web concepts and work with them. Although at first glance one may think that services and agents can be used interchangeably, there are fundamental differences between the two paradigms.

### Differences between Services and Agents

The advent of web technology and the desire to build distributed systems out of existing software components that may exist in different organizations gave rise to the concept of a Service-Oriented Architecture (SOA). The SOA paradigm structures a software system as a collection of services, communicating through a common facility, such as an Enterprise Service Bus (ESB). A service is a software component that performs a specific function and has a well-defined application programming interface (API). The service API defines the functions that are performed by the service and its input and output (Brown 2008).

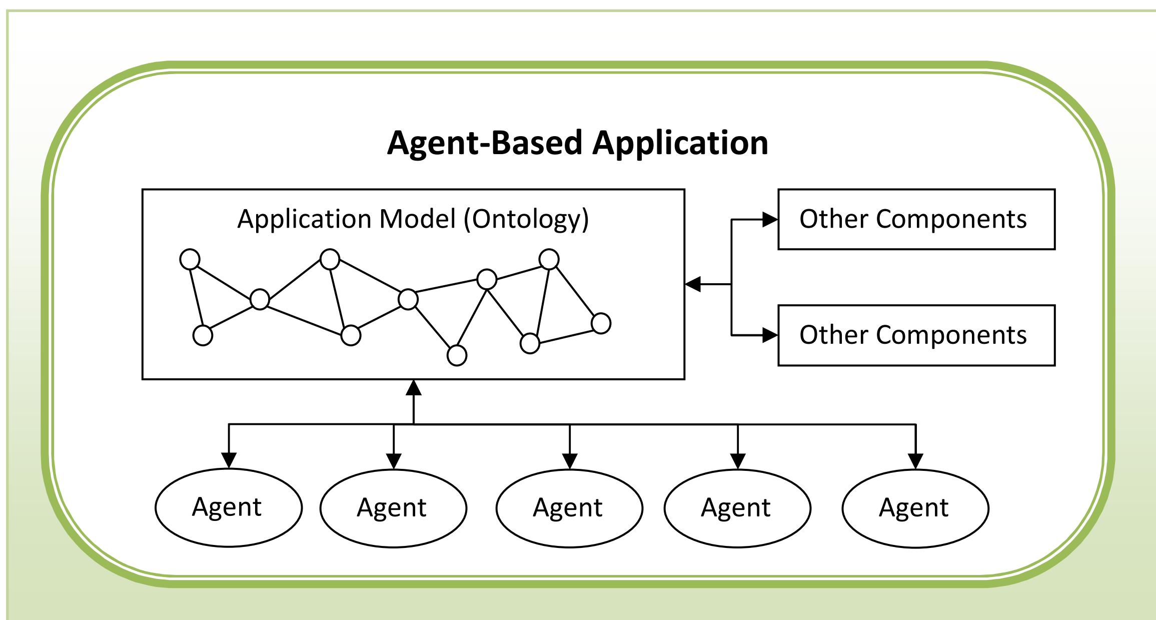


Figure 3: Typical architecture of agent-based systems

Software agents are software components that are situated in an environment and capable of flexible, autonomous action (Figure 3) (Wooldridge & Jennings 1995, Jennings et. al. 1998). Being situated in an environment means that the software component receives sensory information from the environment and can perform acts, which change the environment (e.g., create new objects, delete existing objects, change values of object attributes, or change relationships among objects). Software agents are also autonomous because they can take action without being explicitly invoked by the user. The changes in the environment trigger their action,

based on their interest. Agent autonomy also means that agents have control over their internal state. They determine what information they keep internally to maintain their awareness of the environment and the current state of their interest satisfaction.

The flexibility of agents is described by three qualities: responsive; proactive; and, social. Agents are responsive because their actions are triggered in response to changes in the environment. If the information in their environment is updated and the change affects their interest, they respond to that change directly. They are proactive because they have a set of interests and they try to satisfy them by taking actions based on the available information. The social quality of agents means that agents are capable of communicating with other agents (or human users) by providing information about their current internal state, the degree of satisfaction of their interests, and possibly the reasoning behind any action they take.

There are other qualities that can be bestowed on agents, such as mobility (the ability of agents to move from one server to another and perform functions on every server they move to), learning (the ability of agents to acquire new knowledge and update their current set of interests), and intelligence (the ability of agents to perform analysis on the environment and produce recommendations, alerts, and warnings) (Wooldridge et. al. 1999; Barber et. al. 2003).

The above description of both services and agents shows that the two software components are different in fundamental ways. While agents are embedded in an environment and receive updates about the current state of that environment, services are totally unaware of any environment external to them and have no knowledge about any system that uses them. Agents act proactively without user invocation, while services are explicitly called through a well-defined API. Agents can perform acts to change the environment, while services can only accept input and produce results that are passed onto the calling component, which decides how to use the service.

**Approaches for using agents in SOA.**

The two paradigms, SOA and agent-based systems, are different and serve different purposes. However, they can complement one another. One approach to combining agents with services is to build agent-based services (Figure 4). Such a service can be as simple as a unit conversion software component, or it can be as complex as a planning system with access to external databases or other data sources. In the case of complex applications, agents can be embedded within the service. They understand the internal model of the service and monitor its state. When the internal state of the service changes, agents can react and produce their analysis or cause a change in the service environment.

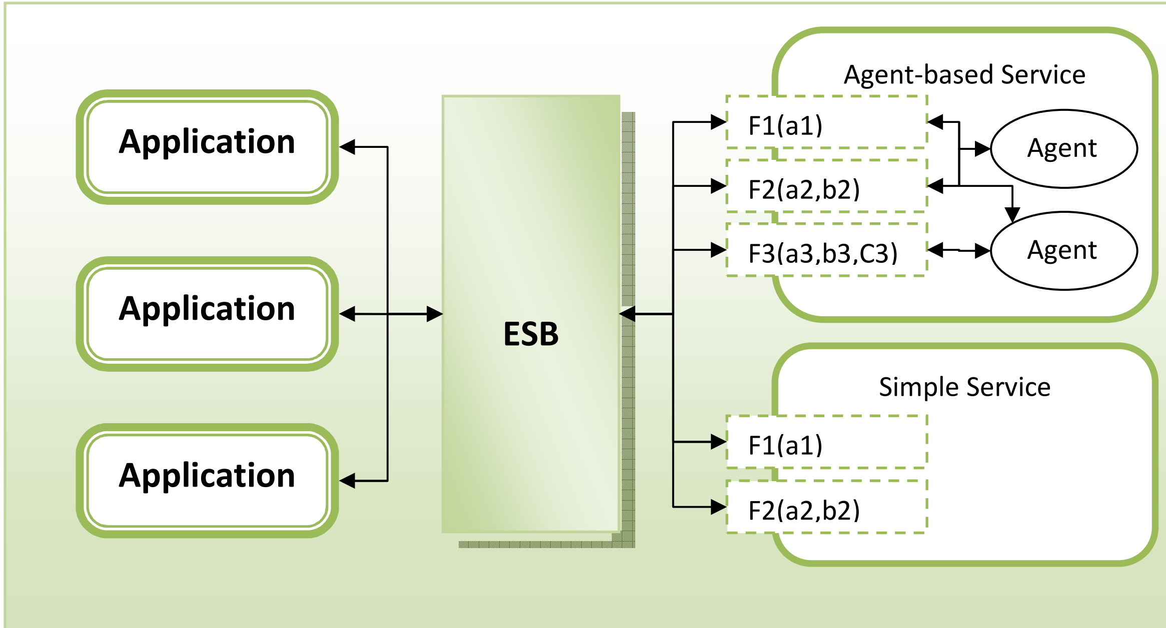


Figure 4: Agents as services

Another approach is to use agents in the main application and use services as external components (Figure 5). The client application will be a service-based application as well as an agent-based application. The application's internal model is the environment in which agents are embedded. In this architecture, agents operate on the application information, by monitoring the current state and by having the ability to take actions to change that state. The agent functions are related to the application objectives. They provide analysis that is related to the application function and may initiate requests for external services. Such requests go through the ESB like any other application request for service.

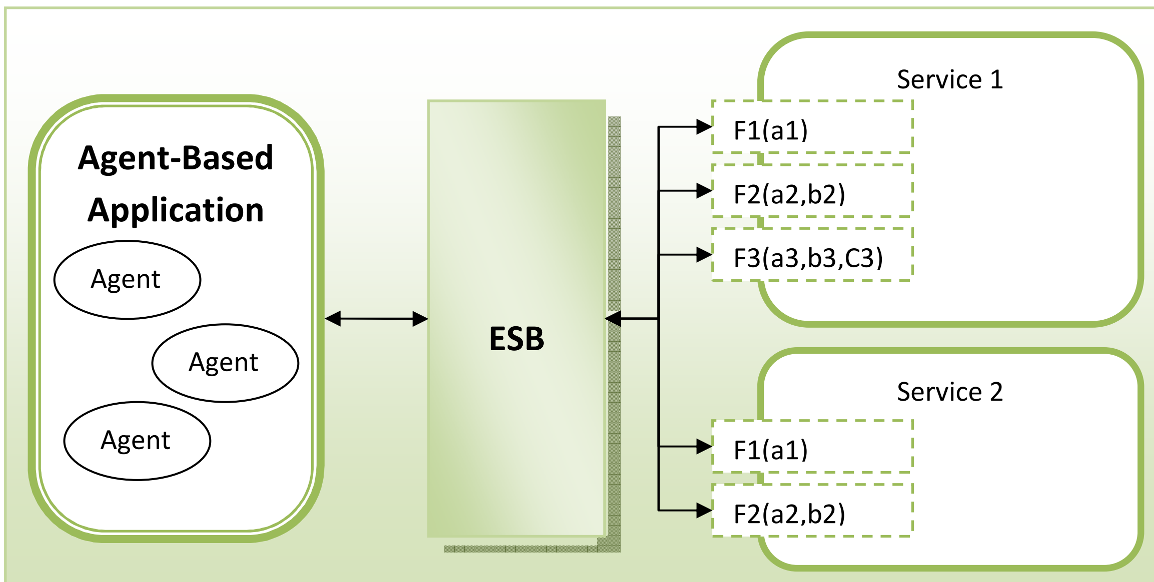


Figure 5: Agents as components that use services

Agents can also play a role within the ESB (Figure 6). Some components of the ESB provide assistance in locating, executing and monitoring services. Other components provide assistance in mediating service requests and data requirements. Intelligent agents can provide help in dealing with such issues, especially when the number of services grows and the constraints on their use and access become more complex.

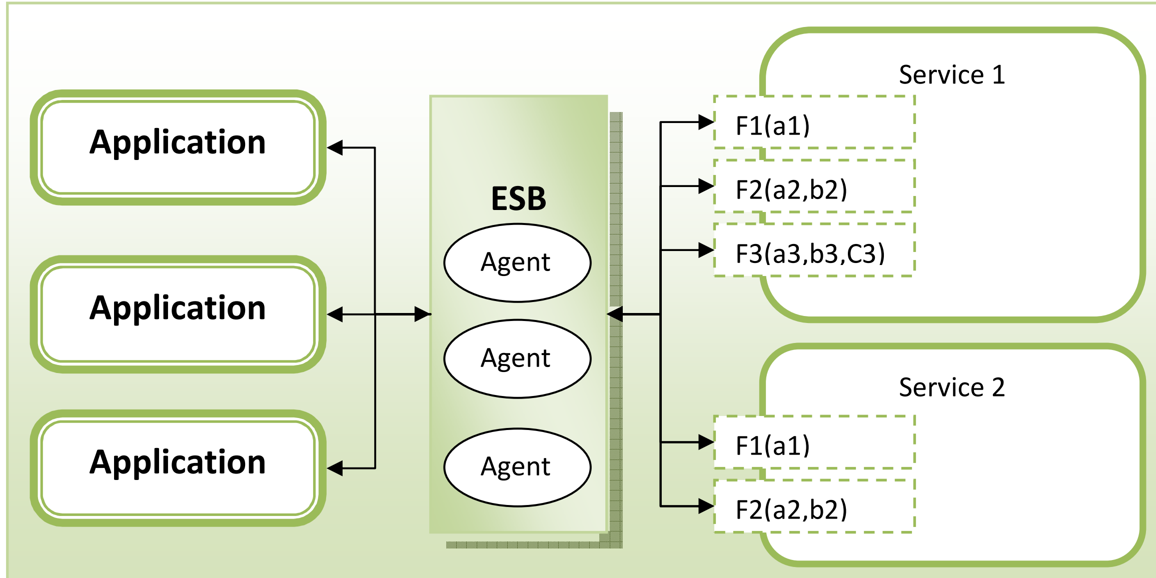


Figure 6: Service management agents

## Recommendations for Using OWL to Build Systems

### The Reasoner

In OWL-based software development the reasoner plays a central role in building the system. It is the component that communicates with the OWL ontology and with the other components of the system. Therefore, the choice of a reasoner must be made carefully in order to ensure that the system components receive the information they need from the ontology in an accurate and timely manner.

It is likely that existing reasoners, such as JENA, may need to be extended to suit the specific requirements of a system. The main function of a reasoner is to determine the consistency of the current state of the ontology given a specific data set. When the data change, it is essential that the consistency check be executed again. This is time consuming and may not serve the purposes of the user. Under these circumstances it may be important to consider a truth maintenance component as an extension of the reasoner. The truth maintenance component would respond to changes in the data set and examines their implications on other associated data items. It makes the necessary adjustments to relevant data items only without examining the entire ontology data set. This capability is typically implemented through a set of rules identifying significant expected changes and specifying the appropriate responses.

It is useful for the reasoner to provide asynchronous communication with the system components, so that large updates may not affect the performance of other system components. Asynchronous communication is necessary when there are many components and each

component subscribes only to a small subset of the information. Updates can be pushed to clients based on their subscription profile.

Reasoner performance may be one of the most important considerations when dealing with a large volume of data. There are reasoners that utilize the Rete algorithm (Forgy 1981), such as Bossam and FuXi. The Rete algorithm was designed to handle a large volume of data with a small number of rules. The basic idea is to build a network of patterns that represent the rule conditions and examine the data as it comes into the network for matches. This algorithm reduces the number of checks that have to be performed and makes the pattern matching more efficient.

### The Database

The back-end database, which holds all the information in the OWL ontology, must be designed to support the following:

- The storage of large volumes of data.
- The storage of all the information that may be implied by the relationships among concepts.
- The ability to retrieve large chunks of data in reasonable time.

The database serves as the main repository of information from the ontology. Some level of inference can take place within the database or at the retrieval stage. SQL queries can be designed to return data that represent a given condition. The returned data are typically a small subset of the database (i.e., the ontology data) and can be further used to perform more complex processing by the querying component. In this view, SQL can be considered a preliminary inference mechanism.

### Building Rules

Agents can be designed as sets of rules that fire based on the satisfaction of their conditions. Such rules can be built manually or, in some cases, automatically.

***Manual Rule Development:*** Rules can be developed as part of the OWL ontology or as a client to another system, and receive their ontology updates through the reasoner. In both cases, the design of agent rules has to consider the following:

- Rules must fire on their own as soon as their conditions are satisfied. They should not require any user interaction to fire.
- Rules can produce additional information, which has to have representation in the ontology. This information may be alerts for the user, requests for other information, or changes to existing ontology objects.
- Agents can be represented in the ontology. This offers the opportunity to associate alerts, or other information types, with the agent that created them, and by doing that, creating the ability of tracing agent results and building justification for agent actions.
- Agent representation also allows the tracing of agent status (e.g., running, idle, has alerts, requires user attention, etc.).

***Automatic Rule Development:*** Some rules can be generated from the current state of the ontology. For example, a set of rules to monitor the status of an organization, based on the current activities or associations of its members. The knowledge needed to create such rules may be embedded in a rule-generating agent. Such an agent monitors the objects of interest in the ontology and creates monitoring agent with the appropriate set of rules and associates it with the observed object. Dynamic rules can be generated in one of the following ways:

- Customizing a generic rule with the relevant ontology objects.
- Assembling rule components (i.e., conditions and actions) from existing representation in the ontology.
- Building rules from scratch, based on knowledge that is embedded in the rule-generating agent.

### Tracing Rule Firing

The ability to trace the firing of rules and chain rule dependencies is supported in backward-chaining rule engines. In such systems, rule firing is triggered by a stated goal and the rule engine attempts to satisfy this goal by firing relevant rules. The engine keeps track of the chain of rules until the goal is achieved or it determines that the goal is unattainable. The rule chain is, typically, accessible to the user.

In forward chaining rule systems, tracing the firing of rules has to be explicitly implemented. The representation of agents in the ontology (see Manual Rule Development) can be expanded to include rules of interests (or all rules). The representation of rules includes status, associated objects, firing order, and so on. In the implementation of agents, every rule updates its representation in the ontology with relevant information. The rule tracing component utilizes this information to analyze the rule firing sequence and the associated information in any desired way and can re-construct a rule firing scenario and possibly extract explanation of agent actions.

The representation of rules should also have a rule description that provides high level explanation of the rule behavior. It is important for the user, when tracing the rule firing, to see what the rule is supposed to do. A rule description can be a simple text field associated with each rule, describing its intended use. It can also be a more complex description, generated from the structure of the rule.

### Visualizing Rules

It is desirable to convey the dynamic nature of agents to the user by including some indication of the activation of agents in the user interface. Agents can have representative icons on the user screen to indicate agent status. Certain rules may also have their own graphic representation on the screen to indicate their status as well. Rule icons can be grouped into their agent icons, which can be maximized or minimized to control screen clutter and to provide better visual experience.

User interaction with agents is possible by expanding the agent graphic representation on the screen from an icon to a window, possibly with text fields to present agent messages to the user and forms to capture user input. Simple interaction may ask the user to acknowledge some alerts or turn off some warnings. More complex interaction may ask the user to guide the agent operation by providing additional information or selecting from multiple courses of actions.

A very important part of this interaction between user and agents is an explanation facility. The more intelligent the capabilities of the software the more important it is for the user to be able to ascertain why an agent has come to a particular conclusion. In the case of rule-based agents an explanation facility can include a tracking mechanism that is built directly into the rules and generates explanations automatically.

## Conclusion

To develop intelligent web applications, two paradigms need to interact effectively. The Service Oriented Architecture (SOA) paradigm offers the structure and interaction management of service-oriented software components operating within a networked environment. The Web Ontology Language (OWL) offers the flexible and powerful modeling and inference capabilities necessary for software to reason over, or otherwise analyze information. Combining the two paradigms in a workable architecture offers great opportunities for developing intelligent web applications that take advantage of the distributed services capabilities as well as sources of information on the Internet. The architecture proposed in this paper provides a hybrid solution that seamlessly marries these two environments via a transactional bridge. The resulting combination supports the inference of web-based content within a service-oriented fabric that is the emerging form of the Web.

## References


Barber K., A. Goel, D. Han, J. Kim, D. Lam, T. Liu, M. MacMahon, C. Martin and R. McKay (2003); 'Infrastructure for Design, Deployment and *Experimentation* of Distributed Agent-based Systems: The Requirements'; The Technologies, and an Example, Autonomous Agents and Multi-Agent Systems. Volume 7, No. 1-2 (pp 49-69).

Berners-Lee, Tim; Hendler, James; Lassila, Ora (2001). "The Semantic Web". *Scientific American*. 17 May.

Bock, Jurgen; Haase, Peter; Ji, Qiu; Volz, Raphael. (2008) Benchmarking OWL Reasoners. In ARea2008 - Workshop on Advancing Reasoning on the Web: Scalability and Commonsense; June.

Brown P. (2008); 'Implementing SOA: Total Architecture in Practice'; Addison-Wesley.

Erl T. (2008); 'SOA: Principles of Service Design'; Prentice Hall.

Forgy C. (1982); 'Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem'; Artificial Intelligence, 19 (pp. 17–37).

Jennings N., K. Sycara and M. Wooldridge (1998); 'A Roadmap of Agent Research and Development'; Autonomous Agents and Multi-Agent Systems, Vol. 1 (pp. 7-38).

McKendrick, Joe. (2009). SOA adoption trends -- what the data tells us. March 12. ZDNet News and Blogs, Service Oriented. http://www.zdnet.com/blog/service-oriented/soa-adoption-trends-what-the-data-tells-us/1679

Wooldridge M. and N. Jennings (1995); 'Intelligent Agents: Theory and Practice'; The Knowledge Engineering Review, Vol. 10(2) (pp. 115-152).

Wooldridge M., N. Jennings and D. Kinny (1999); 'A Methodology for Agent-Oriented Aanalysis and Design'; Proceedings Third International Conference on Autonomous Agents (Agents-99), Seattle, Washington.