# A High Level Service-Based Approach to Software Component Integration

Y. Arafa, C. Boldyreff, A. Tawil, H. Liu
School of Architecture, Computing, & Engineering
University of East London,
London, UK
{y.arafa;c.boldyreff}@uel.ac.uk

*Abstract*— **This paper motivates and sets out a framework for a high-level approach to software component integration. The framework builds on the concept of SaaS (Software as a Service) and uses a service ontology for the annotation of software components with formal specifications. The ontology is used to instruct interoperability between software components through a unified API interface. The impetus for this approach is to provision for smooth integration, management and scalability in a collaborative and distributed development environment.**

***Keywords- Collaborative Software Development, Component Reusability, Software as a Service, Service Ontology Application***

## I. INTRODUCTION

Collaborative development often relies on a modular, component-based approach to software engineering. This approach to system development is by nature open and extensible, and must rigidly support interoperability and data exchange from a diverse range of sources. Opening way to problems of identifying components that provide the required services/functionality and their corresponding data exchange format and interaction pattern. Consequently, developers and incorporating systems must agree shared representations that explicate the service and define interoperability and functionality specifications for successful integration.

Typical component-based integrations inherently experience one or all of the following problems [3, 4, 7, 11, 13]: requirements mismatch and component interoperability issues; difficulty locating components; service/functionality assumption conflicts; overall functional break down caused by the slightest perturbation to API constructs and/or their Inputs - Outputs - Preconditions - Effects (IOPEs); and problematic to scale when additional data and/or constituent components are required, making upgrades and expansion difficult to seamlessly implement; which can lead to maintenance problems as system complexity increases.

It is unlikely that such challenges are overcome without significant improvement to the underlying development environment and infrastructure. The success of the component-based integration process, hence, relies on both functional and non-functional considerations. Development considerations should include the following [5, 3]:

- Well-defined component interrelationships.
- Unified interoperability mechanism between components.
- Common understanding of the data exchanged.
- Common understanding of component functionality and goals.

A promising solution is to make use of ontologies. Ontologies provide a well-founded mechanism for the representation and exchange of structured information [17]. Information about components, their services and their IOPEs can accordingly be formulated in dedicated service ontologies. Ontology-based techniques provide a means to describe, and reason about functionality and how to interact with other software entities regardless of their technical origins. We introduce an ontology-based approach for specifying the goals of components, and the properties relevant for deployment and assembly in diverse applications.

This paper outlines an approach based on a service-oriented model that uses ontologies to formally characterise components and describe their interfaces, specifying possible interconnections between them and provisioning for semantically annotating data objects that may be exchanged. The approach aims to mitigate difficulties arising from component integration and their adaptation into heterogeneous distributed systems.

## II. BACKGROUND

### A. Service-Oriented Architectures (SOA)

Service-oriented architectures are principally designed to resolve the complexity described in the previous section. In a SOA, services are defined by as a set of well-defined interfaces, which are generic in nature; along with a schema for the input required by the service to function correctly and a description of the output produced. The inherent nature of SOAs is that services work with an extensible schema and thereby can cope with various different types of other services that it may interact with [12].

Continued success of SOA-based applications built for the Web has shown encouraging results. Web service ontologies used to describe Web services and their availability have managed to alleviate many of the problems

of service integration [4, 7, 10, 12] by providing formal means for the following [11]:

- Creation of complex, realistic, and scalable networks of component inter-relationships
- Distribution of autonomous controls and monitors
- Dynamic modification of the component execution structure
- Adaptation and evolution of the overall systems using the services.

### *B. Service Oriented Models*

The Service-Oriented Model (SOM) is a relationship model described by the W3C Web Services Architecture Group and is created to explicate the relationship between the services an entity provides and requests. The model is considered the underpinning for developing service architectures [6] and facilitates for loosely coupled software components to be integrated within other software systems.

The fundamental elements in the model are that of goal state (states of some service or resource that is required by other software entities or individuals that may intend to use the service), service ("an abstract resource that represents a capability of performing tasks"), task ("an action or combination of actions that is associated with a desired goal state"), role (defines a set of related tasks carried out and identified by message properties) and agents (which are "capable of and empowered to perform the actions associated with a service on behalf of its owner") [14].

Component-based software engineering and service-oriented software engineering are two of the most widely used engineering paradigms among the current software development community. Despite being developed separately, both paradigms have much in common and bijective concepts are labelled differently [3]. Service-oriented software bears many similarities to traditional software components, in that they are autonomous or semi-autonomous entities with explicit and succinct interfaces that are distinctly designed with reusability and modularity in mind. The distinguishing point is predominantly the requirement for standard communication models, since such services are geared for the web.

The model serves as a useful reference map in considering the elements of service oriented architectures and a rigorous approach to developing them. The mechanisms defined by the SOM offer means to implement the functionality and focus on the service dynamics and substitutability.

### *C. Components as Services*

A typical view of components is that they implement one or more provided interfaces, where an interface is a contract of functional behaviour. In this sense, interfaces provided by components are very similar to service interfaces. This makes components an ideal candidate for implementing services, where a service has a provided interface.

SOMs are actually based on a simple component model [2], where a service encapsulates a coherent set of operations. A central objective of component development is the separation of computation and operability or interoperability. Computational aspects are abstracted by interfaces with well-defined descriptions. These descriptions are made public and formulate the means for interaction between different entities.

## III. A SaaS-Based PANDORA Architecture for Software Integration

Pandora is an advanced training system for crisis management developed by distributed partners. The underlying intelligent system is a complex framework of heterogeneous software components developed by the different project partners that provide system services for planning, training scenario management, multi-media mash-ups and a 3D virtual environment. To achieve the intended overall system function and behaviour, the constituent software components must not exist in isolation. They must be able to communicate and exchange information transparently, irrespective of the technologies used to implement them. Considering components as services provides an independent function or process. The services in SOA are inherently interoperable by design. This intrinsic interoperability builds on the principle of loose coupling among the services which is achieved by virtue of a canonical communication framework that enables implementation of highly standardised service descriptions and message structures.

The Pandora integration architecture builds upon the concept of SaaS (Software as a Service). SaaS refers to an interoperable computing service model in which the software components are offered as services. These components may require interacting with each other in order to accomplish a task and may be simple singletons performing a single function or complex performing a set of related functions. A SaaS-based SOA enable the definition of services in a technology-independent manner, which plays a significant role in enabling the interoperability of service components, making them more robust, flexible and agile [13, 14, 15]. Encapsulating the business logic in a manner that is independent of the technical details will formally capture the essence of the applications and facilitate scalability and reusability in a variety of different contexts [16, 17].

In the Pandora project components are described in terms of the service they provide, the API interface name and the IO parameters and functional conditions required. This information is annotated into a dedicated component ontology. Using ontologies has allowed the dynamic assembly of software and round-up into a mash-up tool.

This approach provides the mechanism to support technology independent software component invocation through the annotation of component services. It has provided leeway for component developers to focus on the functional processes required from the services.

## IV. ONTOLOGY-BASED APPROACH TO INTEGRATION

The impetus for using an Ontology-based approach to integration is to provide a formal means for effectively connecting disparate components and mitigating inevitable problems both at the development phase, as well as possible future system upgrade or expansion phases.

Integrating different components presents integration implementers with problems on two levels: the interoperability of components from different platforms and vendors; and possible application conflicts resulting from integrating them. Consequently, integration demands consistent representations of data exchange, unified interfaces between software entities, and an effective approach that enables integrated components to function across various platforms [1, 8]. It is hence necessary to build an infrastructure for integration, which is based on such robust conceptual models. Our experience within the Pandora project has shown that ontologies are a promising means to achieve these conceptual models, since they can be used to promote common understanding, and they can be used as basis for comprehensive information representation and communication [10,16].

### *A. Service Ontologies*

Ontologies classifying and describing services are called Service Ontologies. OWL-S [14] supplies a core set of ontology concepts for describing the properties and capabilities of Web services in "unambiguous", "computer-interpretable" form. OWL-S mark-up of services is designed to support the automation of Web service operation, which includes: automated Web service discovery, execution, interoperation, composition and execution monitoring [14].

There are existing conceptual models for describing services such as WSMO, WSDL-S, SWSF, SAWSDL. Like OWL-S, these models also address the semantic nature of web service descriptions thereby making an effort to automate the web service life cycle [2].

In OWL-S, and as delineated in Figure 1, a service is described by specifying a function name, the inputs required, the output of the service and its target address for execution. Service ontologies supplement Web service application development by providing the information required to enable automated services discovery, the execution and assembly of composite web service applications. The idea is to annotate services, enabling the automation of the service life cycle.

### *B. OWL-S*

The OWL-S service ontology is classified into three categories: Profile, Model, and Grounding. A service component is actually an instance of the service and is linked to these categories by different properties.

The profile describes the functionality a service can provide and details on the input and output requirements for that service, as well as any preconditions and effects the service may have as constraints. Input specifies the actual input parameters required for invoking the web service successfully; output specifies the outcome produced from the service execution that a requesting client expects and receives; preconditions define the constraints that need to be satisfied for successful execution; and effects describe the state of the service after execution.

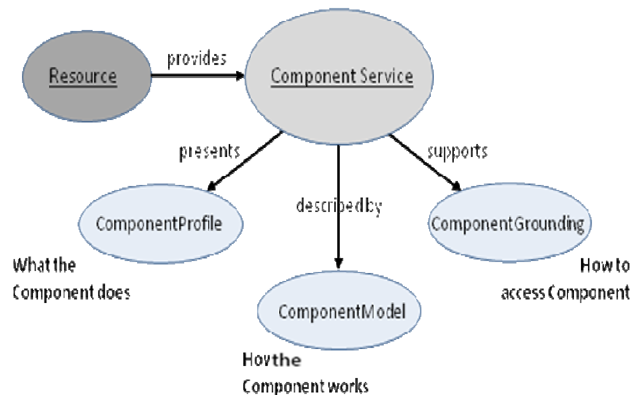


Figure 1. Basis for the Pandora Component Ontology (OWL-S [14]).

The service model describes how the service should work in order to achieve its functionality. It describes atomic processes, composite processes and the message composition involved in invoking the service. Atomic processes are the processes that undergo direct execution requiring a set of specified input parameters; whereas composite processes are those processes that involve the execution of a combination of different services.

Service grounding describes how services are invoked. Grounding defines the network protocols, data and exchange formats needed in order to execute the service successfully.

Although OWL-S is intrinsically developed to define and describe Web-service ontologies, it is suitable to define the Pandora components as services provided even if the end product is not web-based. We have designed the overall Pandora integration architecture as a SOA, considering a SaaS (Software as a Service) solution, where components are services. Therefore, the OWL-S specifications an also apply to component descriptions.

### C. *Component-Service Ontology*

The Component Service Ontology can now loosely be defined in terms of the following along the guidelines set out by W3C in [15]:

- A classification of re-usable components according to the functions they provide.
- A mechanism for rigorously describing, identifying and integrating within heterogeneous systems both during design and at runtime.

Effectively, the developed ontology is a description of the Pandora component APIs and includes the details necessary to invoke and use the implemented components. The components themselves reside in a common repository of services that has been updated throughout the Pandora system development phase, and will be utilised further if future upgrades to the Pandora system occur.

### D. *Component Descriptors*

The following table outlines the descriptors that have been produced for each of the outer-level Pandora components. The descriptors are required in order to compile an ontology of component services. We have included details based on the OWL-S specifications and the technical requirements necessary for integration purposes. Component descriptions are defined Table 1 below.

TABLE I.
COMPONENT DESCRIPTORS

| | |
|---|---|
| Category | Service category the component belongs to. |
| Class | For native lang APIs include class component belongs to |
| Identifier | Component construct or API name |
| Description | Text describing the functionality or service component provides. |
| Author(s) | Partner/ individual developer(s) |
| Version | Number |
| Creation Date | Date |
| Modified Date | Date |
| Location | URL location of component if applicable. Otherwise, local assumed. |
| Interface | Service: executable or Language: Java, C++, C#, |
| OS Platform | Windows, Mac, Linux |
| Input | List of input (if any) require by the component, along with data types. |
| Output | List of output (if any) produced by the component, along with data types. |
| Precondition | List of conditions that should hold prior to the service being invoked. |
| Result Condition | List of statements that should hold true if the service is invoked successfully. e.g. “Package being delivered” |

## V. DESIGNING FOR EXTENDED FUNCTIONALITY AND USE

Development projects are usually produced with the intention of possible future scalability. The integration proposal supports the seamless integration for future extension to a development environment by describing parameters and functionality and a mediator that will manipulate the integration and seamless flow. Continuous collaborative development may require the addition of further components. Using a service-oriented architecture and an ontology-based approach to component descriptions provisions for the semantic integration of software components, we aim to assist in the dynamic assembly of additional components and plug-ins within evolving versions of the Pandora system.

Accordingly, the high-level approach to component integration as adopted in the Pandora system development supports the following:

- Dynamic adaptation across development languages.
- Simple, unified component invocation through the Pandora Mediator.
- Querying component availability and validating the execution process.
- Elegant exception handling.
- Component code changes or evolution and recover accordingly, supporting modular component development.

### A. *An Integration Framework*

The Pandora system architecture uses a component-based design relying on the concept of component decoupling. This is a strong principle of SOAs, which emphasises creating components that are self-contained and have a clear separation of concerns. There is a separation between the function of individual components and the operation of the Pandora system as a whole. Each component has distinct functional behaviour that can be utilised by other components through well-defined interfaces. Component interfaces and behaviour descriptions are advertised in an agreed Pandora service ontology.

The architecture is built as a SaaS solution. The following principles govern the design of the overall integration architecture. They reflect the requirements for service oriented and distributed environments in order to provision for the seamless integration of the Pandora system components:

*Service Oriented*: providing a description of component services that include aspects of communication, structure, and processing logic. This includes service reusability, decoupling, abstraction, autonomy, and discoverability.

*Distributed*: applied to the architecture middleware, which supports the management of possible distributed components transparently such that the execution

process can be scaled across a number of physically disperse servers.

*Semantic annotations*: providing rich and formal description, based on OWL-S, of components and behavioural models defined in the Pandora Component ontology, which enables scalable and seamless interoperation, discovery, reusability assembly.

*B. A Conceptual Overview*

A conceptual view of the integration framework is illustrated in Figure 2. Based on the principles outlined above, the framework uses ontologies for application integration on the component interface level by characterising components, specifying possible interconnections between them and provisioning for semantically annotating data objects that may be exchanged. A component mediator is used to process the ontologies and facilitate interaction between disparate components, thus enabling integration at run-time.

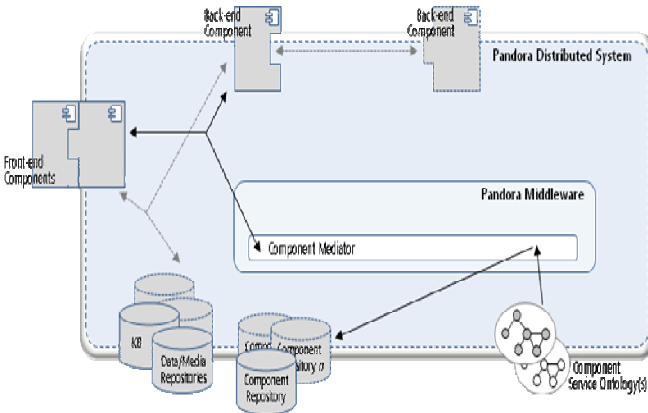


Figure 2. Conceptual view of ontology-based component integration.

Key to this approach is a design pattern that is domain independent facilitated through the definition of a Pandora Middleware. The middleware deals with the transition from static, hardcoded interfaces between components to dynamic interfaces via the middleware. The middleware then becomes a universal interface used by individual components in an overall assembled system. The idea is that components are no longer directly interconnected, but interconnect via the Pandora Middleware, thereby becoming accessible to all in a uniform manner.

The proposed ontology is utilised by the Pandora Middleware to direct component integration. Any changes to component interfaces and/or behaviour descriptions will be maintained in this ontology. The Middleware is able to handle integrity problems if there are conflicts between the advertised component interfaces and those expected by individual components by looking up the ontology. The middleware will degrade overall system functionality gracefully by alerting requesting components to the required format if mismatches occur.

The Component Mediator makes use of specially constructed component adapters and data wrappers, alongside a service ontology to enable software components interoperability through service-sharing.

The Middleware is the core of the architecture providing the main intelligence for integration and interoperation. It consists of a number of components (middleware services) where each component provides certain functionality within an execution process. Each component exposes its functionality through a number of interfaces, thus the functionality of the component could be deployed by other components through these interfaces.

The framework does not mandate that all components must interact via the middleware. Indeed, components of a common category may directly interact with each other without the need for middleware mediation. This may be particularly useful for components that are part of a pre-assembled package. However, scalability of the components within the package cannot be supported by the middleware and must be handled externally.

The components that represent the middleware services include functionality for component discovery, selection, managing interoperability and run-time execution, data and process mediation, exception handling and resource management. In addition, the middleware has been implemented to operate in a distributed manner on a number of physical servers connected using a shared message space. Shared spaces provide a messaging abstraction for distributed architectures, as well as supporting the scalability of the integration process [7].

*C. The Pandora Middleware*

The middleware architecture is an internal communication mechanism for an architecture that relies on an events-based model. The overall architecture of the middleware is structured into four main parts: an Event Bus, a Component Mediator, an Execution Manager and a Resource Manager. This section describes the overall middleware architecture and briefs on how its various components interoperate. Figure 3 shows a high-level depiction of this architecture.

*1) Event Bus*

The Event Bus maintains and manages invoked components and sequences of requests for component services. It combines event driven and service-oriented approaches to request handling and management, so as to facilitate seamless, persistent interoperability of components across heterogeneous platforms.

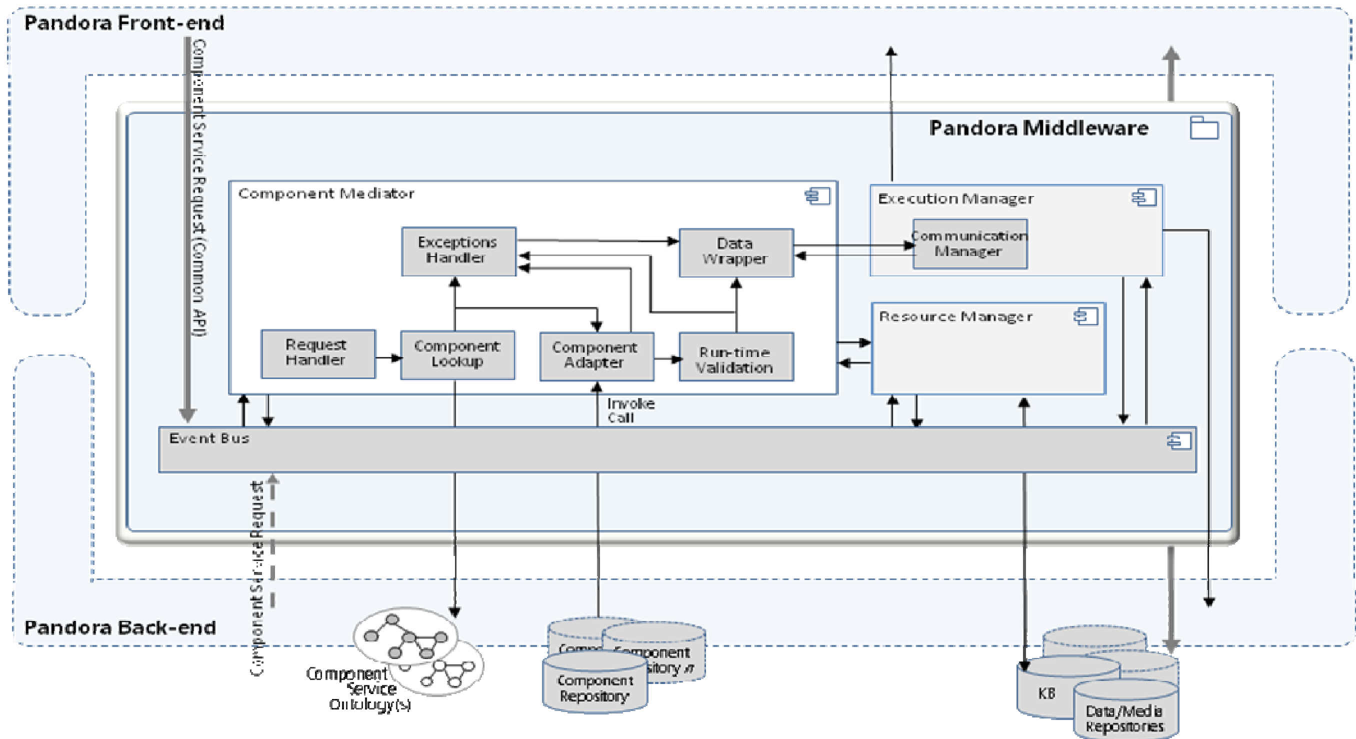


Figure 3. Integration Middleware Architecture

The Event Bus serves as the first-line of interaction for components. Once a request for a service has been initiated, an event is created and placed on the bus with a time stamp, a sequence number, a priority and a process state. It subsequently passes request information to the Component Mediator for appropriate handling; then once invoked successfully, it is placed back on the bus with an available state. Execution and data exchange can now begin. Execution is managed with the support of the Execution and Resource Managers.

Once components are available, their services are accessible to all other requests that can satisfy the defined preconditions. The Event Bus further uses the Component Mediator to enable seamless interoperability between deployed components.

*2) Component Mediator*

The Component Mediator makes use of specially constructed component adapters and data wrappers, alongside a component (service) ontology, to enable component interoperability. Its functionality includes validated component invocation, messaging and data transformation.

The Component Mediator also has the role of reconciling integration conflicts that may occur when trying to invoke a requested component. It can deal with the reconciliation of message exchange patterns on the component descriptions as defined in the Component Ontology.

The Component Mediator consists of the following:

*Request Handler*

Requests are instigated by components requiring particular services at run-time or when initialising the Pandora system. Requests are initially received by the Event Bus and passed on to the Component Mediator where the Request Handler initiates the process of validation and invoking requested services.

Requests are made through a unified API call to the Event Bus that includes the necessary parameters that satisfy a service's IOPE requirements.

*Component Lookup*

This provides a component discovery mechanism that seeks to find the required component service description(s) that match the goal specified by the requester. It returns a best-match component service that satisfies the supplied requirements. It may return a "not found" string indicating the requested service component is either not available or that the information supplied is not correct, in which case the results pass through the Exceptions Handler for elegant system error handling.

Future versions of the system may provide more than one component that provides the same service. The

difference in this case would be their IOPE and possibly other processing variations. In this situation a list of component services will be returned. This necessitates the introduction of a new middleware component capable of intelligently selecting the most appropriate service for the requester. However, at this stage and for the purposes of the current Pandora system integration requirements, only the aforementioned functionality will be supported.

*Component Adapter*

The Component Adapter is used to map an abstract interface to another object, which has the required functional role, but a different interface. An example of use is to one component that uses Java.

*Runtime Validator*

The Runtime Validator makes sure that a requested components service is available, that it can be located at the URL specified in the Component Ontology, that all the IOPEs are satisfied correctly and that the component itself is executed and is running correctly.

*Data Wrapper*

The Data Wrapper helps in data heterogeneity problems that may occur during the lifecycle of all component interaction at runtime. The Wrapper transforms instances of input and output to and from services to the format expected by each component service. It does so by wrapping data into the format required as defined in the Language specification described in the Component Ontology.

*Exceptions Handler*

The Exceptions Handler has the simple role of elegantly capturing errors that may arise from any component and sending back an error message wrapped in the appropriate format to the service requester.

#### 3) Execution Manager

The Execution Manager is responsible for the intelligent routing in order to reliably connect and coordinate the interaction of services across components and maintain transactional integrity. It supports the Event Bus in handling the events and controlling a complex series of interrelated events. It consists of a Communications Handler that is responsible for inter-component messaging that enables message exchanges among component/event services.

#### 4) Resource Manager

Initially, the Resource Manager handles the repositories of components and ontologies, which are in persistent storage. It is responsible for proving an interface for querying and storing to the database storage used by the middleware. Future implementations will handle other resources required by the Pandora system.

## VI. Conclusion

The challenges of building complex software systems are not only in the construction of their software components and the engines necessary for effective functionality, but also in the integration of these components in a smooth and scalable manner. This paper has presented an integration approach adopted by the Pandora project based on a common, unified API interface that utilises the Pandora middleware functionality to handle distributed component interoperability. This allows for components to seamlessly interact and exchange common knowledge spaces and data. The service oriented approach to the design and the implementation of the middleware on the Pandora project has facilitated integration flexibility and system scalability throughout its development. The approach assisted in the dynamic assembly of additional components and plug-ins within evolving versions of the Pandora system.

The proposed integration framework described in this paper can be applied to evolving developments in any software system integration activity. The principles that governed the design of the Pandora middleware technology architecture for integration are as follows:

*Scalability:* The architecture provides a baseline to support future functionality growth requirements. The architecture is able to support scale both horizontally and vertically in order to meet future Pandora system requirements as needed.

*Modularity:* The architecture establishes the building blocks on which future components can be added.

*Minimised Usage Complexity:* The framework introduces a unified API to the middleware so that all components interact with each other in a uniform manner. The middleware is able to execute correctly by looking-up the component services descriptions in the Pandora Component Ontology and verifying the service IOPEs accordingly.

*Shareable (Open Source):* The middleware is provided as an open source tool and is developed using platform independent technology.

*Based on Common Standards:* The solution uses the W3C OWL-S [14] for the Pandora component ontology development, and standard java libraries for the middleware functionality development.

## Acknowledgment

This paper is a product of research and development on the Pandora project. Pandora FP7-ICT-2007-1- 225387 is co-funded by the European Commission under the mixed call on ICT and Security.

## References

[1] Athanasiadis, I., Rizzoli, A. and Janssen, S. (2009) “Ontology for Seamless Integration of Agricultural Data and Models.” 3rd Intl Conf on Metadata and Semantics Research (MTSR’09): 282-293.

[2] Booth, D., H. Haas, et al. (2004). "Web Services Architecture - W3C Working Group Note 11 February 2004." Online: http://www.w3.org/2002/ws/arch/.

[3] Breivold, H., Larsson, M. and Vasteras, R. (2007). "Component-Based and Service-Oriented Software Engineering: Key Concepts and Principles". 33rd EUROMICRO Conference on Software Engineering and Advanced Applications: 13-20.

[4] Bukhres, O. and C. J. Crawley (1996). "Failure Handling in Transactional Workflows Utilizing CORBA 2.0" 10th ERCIM Database Research Group Workshop on Heterogeneous Information Management.

[5] Castro, R., P´erez, A., Garc´ıa, O. and Nixon, N. (2008) "Towards a component-based framework for developing Semantic Web applications" ASWC, Lecture Notes in Computer Science (5367/2008):197-211.

[6] Dong, J., Sun, Y. and Yang, S. (2005). "OWL-S Ontology Framework Extension for Dynamic Web Service Composition" " 11th International Conference on Software Engineering & Knowledge Engineering (SEKE 06}: 544-549.

[7] Hogg, T. and Huberman, B. (1991). "Controlling Chaos in Distributed Systems." IEEE Transactions on Systems Management and Cybernetics 21: 1325-1332.

[8] Falbo, R., Guizzardi, G., Duarte, K., and Natali, A. (2002): "Developing Software for and with Reuse: An Ontological Approach," Conference on Computer Science, Software Engineering, Information Technology, e-business and Applications (CSITeA-02):477-488.

[9] Iqbal, A., Ureche, O., Hausenblas, M. and Tummarello, G. (2009) "LD2SD: Linked Data Driven Software Development", 21st International Conference on Software Engineering and Knowledge Engineering (SEKE 09): 240-245.

[10] Jin, D. and Cordy, J. (2005) "Ontology-Based Software Analysis and Reengineering Tool Integration: The OASIS Service-Sharing Methodology." 21st IEEE International Conference on Software Maintenance (ICSM'05): 613-616.

[11] Fowler, M. (2003). "Components and the World of Chaos." IEEE Software 3(3): 83-85.

[12] Mandell, D. and McIlraith, S. (2003): "A Bottom-Up Approach to Automating Web Service Discovery, Customization, and Semantic Translation," 12th International World Wide Web Conference Workshop on E-Services and the Semantic Web (ESSW '03): 89-96.

[13] Madiajagan, M. and Vijayakumar, B. (2006): "Interoperability in Component Based Software Development." World Academy of Science, Engineering and Technology (22): 68-76.

[14] OWL-S, online: http://www.ai.sri.com/daml/services/owl-s/1.2/

[15] W3C, Ontology Driven Architectures and Potential Uses of the Semantic Web in Systems and Software Engineering, 2006, online: http://www.w3.org/2001/sw/BestPractices/SE/ODA/.

[16] Wallace, J. and Hannibal, B. (2008). "Software and Hardware System Integration and Intelligent Automation using Ontology-based Knowledge Representation Technology," International Conference on Artificial Intelligence, World Academy of Sc. (IC-AI08):475-483.

[17] Ye, J., Coyle, L., Dobson, S. and Nixon, P. (2007). Ontology-based Models in Pervasive Computing Systems. Knowledge Engineering Review 22(04), pp. 315-347.