

A Model for Service Oriented Computing

Arun Kumar, Anindya Neogi
IBM India Research Laboratory
New Delhi-110016, INDIA
Email: {kkarun, nanindya}@in.ibm.com

D. Janaki Ram
Dept. of CS&E, IIT Madras,
Chennai-600036, INDIA
Email: d.janakiram@cs.iitm.ernet.in

Abstract

The need for having interoperable systems despite the presence of heterogeneous platforms has resulted in tremendous growth and acceptance of Web Services in recent years. This success of Web Services technology is fueling research efforts towards generalization into a science for Service Oriented Computing (SOC). However, beyond the basic publish-find-bind model there is not much consensus yet on what are the architectural building blocks of a Service Oriented Architecture (SOA). In this paper, we present a model for SOC. The proposed model leverages established and proven techniques to define some of the software engineering principles for building an SOA.

Keywords: SOA, Web services, Semantic Web, Object orientation, grid services, composition

1 Introduction

Various programming models have been proposed in the past for building distributed systems. Among these some of the popular ones included CORBA, DCOM, and Java RMI etc. While DCOM and RMI lacked interoperability, CORBA was considered a heavyweight platform. Moreover, synchronous Remote Procedure Calls (RPC) was the predominant style of communication provided by them.

Recently, Web Services [7] have gained acceptance as the technology of choice for building distributed systems. They are being visualized as the most promising enabler for business process integration. They separate end point interface from their implementation and also, enable loose coupling between participating entities. A standardized message format (SOAP [27]) and a standard description language for specifying end point interface (WSDL [31]) establishes the basic protocol for

interaction. XML is used as a vehicle for transporting messages in a platform independent manner.

While the distributed programming community developed a standard message format and the interface description language, the semantic web community approached from a different angle. They used ontologies and AI concepts to introduce semantics into web service descriptions resulting in languages such as OWL-S [22]. The focus there has been to enable automatic web service discovery, invocation, composition and monitoring through programs that can reason.

Grid computing community¹ [1], and others [23, 17, 5] recognised the need for various management functions such as lifecycle management, fault handling, accounting, high availability, support for stateful web services, monitoring etc., that would be required for most pragmatic, web services based applications. As a result, either extensions to existing Web Services architecture [7] have been proposed or frameworks such as Open Grid Services Architecture (OGSA) [14], have been developed on top of Web Services.

Together these different web services technologies are expected to define the future of World Wide Web [30]. In order to fulfil the promise that web services hold and to synergise the efforts going on in different communities, research is underway towards generalization into a science for Service Oriented Computing (SOC). However, beyond the basic *publish-find-bind* architecture there is not yet much consensus as to what exactly should be the engineering principles of a Service Oriented Architecture (SOA). This is due to absence of a normative model² that defines the basic axioms of SOC using which SOA systems can be built.

In this paper, we present a model for Service Oriented

¹<http://www.gridforum.org>

²Recently, OASIS SOA TC (http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm) was formed to develop a reference model for SOA.

Computing that is based upon fundamentals of well established object oriented (OO) paradigm. This choice is despite the fact that there is a lot of disagreement over the issue whether services are similar to objects [29, 10, 21, 13]. In most cases, the differences cited are related to system level issues such as mode of communication supported (synchronous/asynchronous) or the protocols involved etc. However, since services in SOAs are meant to be business level entities rather than fine-grained objects, there are bigger architectural issues that need to be considered. These are concerned with providing internet-level scalability, handling cross-enterprise involvement, maximizing software reuse, and handling normal evolution while building adaptable and robust enterprise systems. Indeed researchers treating Web Services as different from distributed objects [29], refer to them as a 'most widely and popular object oriented architecture' [5]. In that respect, the need and lack of support for OO principles in SOAs has been felt [4, 10, 34, 13] in the community.

In this paper, we illustrate how the basic concepts of OO paradigm such as interface types, instances, inheritance, composition, and polymorphism etc. can be applied effectively to SOC. The model thus created, leverages the rich experience of building successful enterprise integration systems and lays down the basic principles for engineering software in a Service Oriented Computing environment.

The rest of the paper is organized as follows. Section 2 motivates the need for OO principles in SOC. Section 3 briefly present basics of different technologies involved. Section 4 presents details of the proposed model for Service Oriented Computing. Finally, we conclude in section 5.

2 Motivation

Services have been compared to objects using the context of CORBA for distributed object systems and Web Services for SOAs [29, 4, 21, 13]. Implied by this context the primary differences identified are at system level. These include issues such as the wire protocol being binary or XML based, communication mechanism being message oriented or RPC based, service invocation mechanism being synchronous or asynchronous, notion of factories, etc. Some of these are not real differences since they are supported directly or indirectly in both the architectures, though one mode may be preferred over the other [19, 4]. However, since Web Service technologies evolved to enable distributed systems integration while avoiding the pitfalls of previous

approaches, it is natural to have differences in the way these systems are engineered.

The other set of comparisons done is at the architectural level [4, 10, 11]. These are related to representation of state, notion of interface types, service granularity, service reference, lifecycle management etc. In the light of these comparisons we point out that *object based* systems (distributed or otherwise) are not always *object oriented* [16]. To clarify, object based systems allow real world modeling and enable some amount of software reuse through the concepts of class, objects and interfaces. Object oriented systems, in addition, utilize the principles of inheritance and polymorphism [28] to enable substantial benefits in terms of reuse, software maintenance and structured evolution. Distributed object systems need not be object oriented as they basically provide communication between program-level objects residing in different address spaces. They attempt to provide the programmer an illusion of invoking a method on a local object, when in fact the invocation may act on a remote object [15].

Several researchers have either extended or felt the need of applying some of these OO concepts that are currently missing from SOC [13, 4, 10, 23, 34]. We illustrate the need for OO principles through an example.

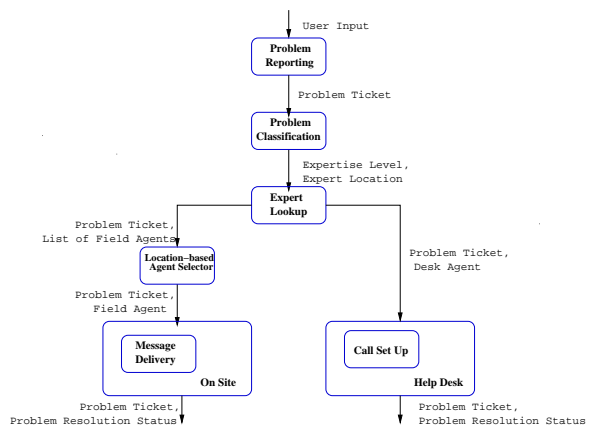


Figure 1. Helpline Scenario

Consider a telecom provider that wants to automate a typical Helpline (or call center) and enable it for one of its clients – a washing machine manufacturer. The helpline workflow operates as follows. A customer calls in to report a problem with her washing machine. This problem needs to be assigned to an agent for resolution. If the problem is such that it could be solved over the phone, a desk-based agent at the call center will be assigned. Otherwise, an agent in the field who can visit the customer and fix the washing machine, needs to be

found and assigned. Figure 1 shows various services involved in realizing this workflow. It is represented in languages such as BPEL³.

Once deployed in a Web Services based system, there are several problems that can occur. For instance, a particular component service such as *Location Based Agent Selector (LBAS)* that selects a field agent based upon her location obtained through mobile phone tracking, might fail. Here, an alternate service will have to be searched from *all* services advertised in the registry even though only a small percentage of those services would be potential candidates. This is because the current Web Services model treats each service description as a representation of a single, running web service and provides no support for grouping. This poses a *scalability* problem.

Furthermore, the client program(s) using LBAS need to change since the Web Service to be invoked is specified at design time. Even though mechanisms such Web Service Invocation Framework (WSIF) [12] allow dynamic invocation and late binding but the interface needs to remain the same. In our case, the available alternate service might provide enhanced functionality such as using Global Positioning System (GPS) in addition to mobile phone tracking. This means that it may have additional operations in its interface.

Similarly, if another client signs up for helpline workflow then some components may have to be customized. For instance, the *Expert Lookup* service might have to change to support the search criteria or lookup mechanism preferred by this client. Therefore, multiple variants of a service need to exist simultaneously resulting in multiple versions of the helpline workflow to coexist. Such *adaptability* is currently not supported but is essential to allow systematic evolution. In this paper, we illustrate how these problems get resolved by employing an OO approach to building SOAs.

3 Background

In this section, we highlight some key features of various technologies that play an important role in the formation of the proposed model.

3.1 SOA and Web Services

Service Oriented Computing paradigm proposes to utilize self-describing, platform-agnostic computational elements as fundamental components for developing

distributed software systems [23]. Such components are modeled as technology neutral and loosely coupled services with well defined interfaces, that can be invoked remotely in a location transparent manner. As shown in figure 2, various interactions can then be captured in a publish-find-bind model involving a service client, service registry and a service provider.

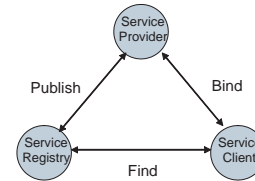


Figure 2. Service Oriented Architecture

The concept of Web services is a particular realization of an SOA the key elements include [9, 29]:

- An encapsulated application that exports a well defined interface using XML-based Web Service Description Language (WSDL) [31].
- A standardized messaging protocol such as SOAP, for interoperable information exchange.
- A service address which is essentially a protocol binding combined with a network address that a requester can use to access the service.

3.2 Semantic Web Services

Web services are meant to be accessed and invoked programatically by software agents. This brings forth an important issue of establishing semantics of the information exchanged. The main aim of Semantic Web Services community⁴ has been to provide semantic annotations to web service descriptions so that they become programatically discoverable, invocable, composable as well as monitorable.

A web service is annotated with Input, Output, Precondition and Effect descriptions ($\langle I, O, P, E \rangle$) for operations in its interface. Inputs and Outputs describe the data elements flowing in and out of the service. A precondition describes conditions under which the service can be invoked whereas an effect describes the facts that become true after the execution of that service. Web service capabilities are formally represented in ontologies like OWL-S and WSMO.

³http://www.oasis-open.org/committees/documents.php?wg_abbrev=wsbpel ⁴<http://www.w3.org/2002/ws/swsig/>

3.3 Grid Services

A Grid Service [14] is a Web Service that conforms to a set of conventions for such purposes as service lifetime management, inspection and notification of service state changes, and also handling of faults. Grid Services address the need for management of distributed and long lived state that is required in distributed applications.

To avoid pushing the burden to client application, the correlation data is typically maintained across requests using features of the underlying communication protocol, such as cookies with HTTP. WS-Resource Framework [32] clearly distinguishes between the concept of a "service" and the stateful entities acted upon by the service. It proposes an Implied Resource pattern for correlation of state data across multiple requests. This pattern allows service providers to make use of extensibility features in the WS-Addressing [2] protocol to reduce the client burden, in a transport protocol independent manner and without requiring support in the Web services client runtime.

3.4 Object Orientation

OO model has been credited with enabling intuitive modeling of real world entities in a unified analysis and design framework, while providing reuse and maintainability characteristics to the resulting software [16]. For this purpose, it supports each of the important abstraction principles: (1) classification/ instantiation, (2) aggregation/ decomposition, (3) generalization/ specialization and (4) grouping/ individualization [28].

In OO, *object* is a key concept that encapsulates data and behaviour of an entity. Classification – the most important abstraction principle of grouping like things together – is used to define a *class* as a collection of like objects. Objects thus are instances of a class. An object is uniquely identified by its object reference. The behaviour of every object is defined by the interface that it exposes to the external world. The interface contains methods that can be invoked by passing messages to the object. Each object maintains state across method invocations. The resulting behaviour is governed by the message received as well as the internal state of the object at the time of receiving the message.

Composition is another abstraction principle that treats a collection of entities as a higher-level entity. OO systems enable composition by allowing objects to be treated as components of other objects.

The third abstraction principle is Specialization. It allows new entities to be defined by adding some, more

specific properties to existing entities. Specialization relationship when modeled at the code level is called *Implementation inheritance*. It allows definition of new classes (called *subclass*) by reusing existing classes. At a software design level, specialization lets the derived class (called as *subtype*) inherit the interface of the base class and make additions to it. It is called *Interface inheritance*. Since an object of a subtype can substitute an object of the base class, Interface Inheritance enables polymorphic behaviour wherein a single message can be interpreted differently by different objects.

The fourth abstraction principle is Grouping which allows heterogenous objects to be grouped together. OO systems provide this in the form of constructs such as lists, etc. A benefit provided by these abstractions is that the the system can undergo various kinds of changes without affecting the rest of the system.

4 A Model for SOC

Here, we present a model for SOC that leverages proven software architectural principles of OO paradigm as well as the work done in the context of Web Services.

4.1 Classification

The notion of common interfaces is de-emphasized in SOA for increased flexibility and dynamism. However, it has been argued that interface types are actually more important for SOA than DOA [4]. They enable ease of composition and interface specialization while their absence leads to increased cost of integration that nullifies the perceived benefits.

In OO, a class captures the notion of an interface along with state data and all classes are specified in a controlled manner. However, in SOA service definitions would be developed independently, by various service providers and a mechanism is needed to prevent or deal with any resulting ambiguity. A programming model similar to the one employed in the Java Community Process⁵ is needed here where the services community can decide what service definitions are needed and can provide reference implementations as well. To capture service definitions, we utilize the work done in Semantic Web Services [22, 26, 18] as well as in automated service composition [3] to define *Service Type* as:

a semantic description of the service. The description consists of one or more profileTypes and a stateDescription.

⁵<http://www.jcp.org>

A *profileType* consists of one or more *operationDescriptions*. Each *operationDescription* is a 4-tuple $\langle I, O, P, E \rangle$. I and O are sets of concepts from an *ontology* that represent data elements accepted by the service during invocation and made available after the invocation of this operation respectively. Address, *PersonName* and *Organization* are some examples of such concepts. P is the set of conditions that should be true for this operation to be invoked. E is a set of expressions that become true after the invocation of this operation. These expressions typically involving elements from I, O and the *stateDescription*. Together IOPEs of an operation capture the functionality provided by that operation of the service. Most pragmatic distributed applications would require maintenance of state [13, 24] and *stateDescription* defines the state maintained by a service. It consists of a list of *stateElements* each of which is an element from the concept ontology and they collectively represent the state that is maintained.

Having established the notion of a 'class' for SOC, we define the concept of 'instances'. A *Service Instance* represents a running service and is defined as:

an operational description of the service. It consists of a serviceTypeId, one or more profiles, state and a grounding.

The *serviceTypeId* is the reference of the Service Type of which this is an instance. A profile is nothing but the description of an interface of the service. It conforms to a *profileType* of the corresponding Service Type and consists of one or more *operations*.

Each operation is a 3-tuple $\langle IM, OM, NFC \rangle$. IM is an input message consisting of one or more $\langle inputType, dataType \rangle$ pairs and OM is a set of output messages each consisting of one or more $\langle outputType, dataType \rangle$ pairs. The *inputType* and *outputType* are the input and output concepts respectively, defined in the corresponding *operationDescription*. The *dataType* comes from a *dataType ontology* (such as XSD⁶ datatypes). For *PersonName* *inputType*, the *dataType* might be String, for example. NFC is a set of non-functional capabilities of that operation and is a set of $\langle attribute, operator, value \rangle$ triples. The *attribute* comes from the concept ontology and along with *operator* and *value*, it represents the capability offered by that operation of the service. For instance, a non-functional capability could be "responseTime lessThan 10msec".

The notion of a stateful resource and web service acting upon it [13] proposed in WSRF, is similar to the concept of state in objects. Inspired from that, State of

⁶www.w3.org/XML/Schema

a *Service Instance* is defined as a list of $\langle stateElement, stateElementReference \rangle$ pairs where the *stateElement* comes from the *stateDescription* of the corresponding Service Type and *stateElementReference* captures the details of the stateful resource behind the service. *Grounding*, on the other hand, captures the details of protocols supported by the service and the address at which the service can be accessed. It consists of one or more *bindings* and a *serviceReference*. The *serviceReference* may capture more than one address at which the service is available.

The specific methods by which instance level details such as *stateElementReference*, *dataTypes* in IM and OM etc. are specified would vary from one particular SOA to another. For instance, in WSDL 'types' are the realization of a *dataType ontology* and state is captured in grid services by WS-Resource concept [32].

Discussion:

Classification provides several advantages:

- Service Type defines expected behaviour whereas a Service Instance description captures the details of a particular running service. This cleanly separates the functional and non-functional descriptions of a service.
- Service Types provide design time availability of type information enabling client programs to be written irrespective of any running service or service provider. Classification, therefore, enables late binding.
- Programming to an interface masks the client from service failures when alternative instances are available. In the helpline scenario, the failure of Message Delivery service would not require creating a new workflow but switching to another instance of Message Delivery service.
- Classification enables scalable service discovery. Search does not require evaluating interfaces of all advertised services but is only limited to a small set of instances that conform to the desired Service Type.
- Classification helps define "equivalence" operator for services. Two services can be tested as equivalent or not based upon their Service Type information.
- Since Service Type semantically captures the behavioural specification of the service, it can be used to reason about the runtime behaviour of a service instance.

The issues to be addressed are:

- Instantiation of objects in OO systems is done in a controlled manner and all instances always conform to their class definition. In SOC, external programs

would be required to test (and certify) compliance of a service by verifying its observed behaviour against that specified in the corresponding Service Type.

– The Service Types define I/O parameters from a concept ontology and Service Instances map them to elements from datatype ontology. This requires a mechanism to ensure that all references to an I/O concept in a Service Type always correspond to the same datatype in I/O parameters of all its Service Instances. Lack of such a mechanism implies that instances may not be replaceable even though they may be functionally equivalent. An alternative would be use datatype conversion mechanisms while switching to another instance.

– An important contribution of OO was the Object Oriented Analysis and Design [6] technique for designing enterprise systems. A similar analysis and design technique is required for SOC. It would help architects and designers to identify parameters such as right level of granularity for a service and the right set of interfaces to export etc.

4.2 Service Composition

Aggregation or composition is the abstraction principle that enables creating new systems from existing components. Advantages of composition are well understood and accepted. Given a set of component Service Types ST_1, ST_2, \dots, ST_n , a composite Service Type $CT_{1..n}$ can be composed from them using the following axioms:

- State maintained by a composite service is a union of the state maintained by its component services. Formally, $SD_{CT} = SD_1 \cup SD_2 \cup \dots \cup SD_n$, where $SD_{CT} = stateDescription(CT_{1..n})$ and $SD_i = stateDescription(ST_i)$.
- each operationDescription OD_{CT} of each profile-Type of $CT_{1..n}$, is created from those operationDescriptions $OD_j \in ST_j, j = 1..n$, that get composed to realize it.
- if i^{th} operation of the composite service is composed from p^{th} operation of ST_x, q^{th} operation of ST_y and r^{th} operation of $ST_z, x, y, z = 1..n$ then the workflow that realizes i^{th} operation is $OD_{CT}^i = OD_x^p \oplus OD_y^q \oplus OD_z^r$, where ' \oplus ' stands for composition. IOPEs of OD_{CT}^i are composed as given below:

$$- I_{CT} = \{e \mid e \in I_j \wedge e \notin O_k, j, k = p, q, r\}, \text{ where}$$

$(I_j \in OD_j) \wedge (O_k \in OD_k) \text{ and } (k \prec j)$ in the workflow that realizes $CT_{1..n}$. This means that the set of input elements of OD_{CT}^i consists of input elements of all the operations involved *minus* those which occur as output elements of a preceding operation in the workflow.

- $O_{CT} = \{m \mid m \in O_j, j, k = p, q, r\}$, i.e. the set of output elements of OD_{CT}^i is the collection of output elements that become available through operations invoked in the workflow.
- $P_{CT} = \{p \mid (p \in P_j) \wedge (\exists q, q \models E_k), j, k = p, q, r\}$, where $(P_j \in OD_j) \wedge (E_k \in OD_k) \wedge (k \prec j)$ in the workflow that realizes $CT_{1..n}$. This means that the set of preconditions of OD_{CT}^i consists of all the preconditions of the operations involved *minus* those which get satisfied by effects of a preceding operation in the workflow.
- $E_{CT} = \{t \mid t \in E_j, j, k = p, q, r\}$, i.e. the set of effects of OD_{CT}^i is the collection of effects of operations invoked in the workflow.

Often it might be required that the composite service add some local functionality as well as state to the resulting composition. This may be needed to fill in some gap not addressed by the existing services. To incorporate this view, the description of a composite service given above, can be modified to include optional additions to state and operations by the composite service.

Given a set of Service Instances SI_1, \dots, SI_n selected to instantiate $CT_{1..n}$, the non-functional characteristics of composite Service Instance $CI_{1..n}$ are an aggregate of non-functional capabilities of its component Service Instances. Determining aggregate non-functional characteristics from components or vice versa has been addressed in [33, 8].

Discussion:

The advantages provided by an OO approach to composition are as follows:

– Composition in OO systems is done at design time, whereas in current SOAs it is typically achieved at runtime [11] since composition is applied on descriptions of advertised running instances. This requires dynamic service discovery from potentially very huge registries as well as keeping pace with dynamic updates such as failed services or new ones becoming available.

Automated service composition has been studied as a possible solution to handle this problem [25].

Using Classification, composition can be done partially at design time. This could be done similar to J2EE programming model where a programmer uses a large but structured library of classes to create new classes. Since Service Types are relatively static entities⁷, they could be distributed without requiring frequent updates. Such Service Type registries could become part of service creation tooling available to service developers at design time. The runtime aspect of composition is then restricted to selecting appropriate running instances corresponding to Service Types in the partial composition [3]. Selection criteria could involve non-functional capabilities of advertised instances and could be specified in a deployment descriptor.

– Given the specification of a desired service (in terms of its IOPEs), AI Planning techniques are used to verify whether the service is composable from existing component services and how [26]. This not only enables automated composition but could also enable re-engineering of existing systems for migration to SOC. OO approach to composition enables this re-engineering to be done without having to deploy component services first, since conceptual composition can be done using Services Types alone.

The concerns to be addressed are as follows:

– The issue of mapping I/O parameters in the Service Type to datatypes in Service Instances surfaces again here. Since data elements would flow from one service to another in a composition it is important to maintain consistency across various services involved. This means that instance selection criteria also needs to ensure that all services referring to a common I/O parameter in their corresponding Service Types use a common datatype to represent it in their Service Instances. For instance, in the helpline scenario, if Agent’s mobile number in Expert Lookup service is represented using a String then it should be represented using a String, in the Message Delivery service as well. Alternatively, datatype conversion mechanisms could be used. Some other data flow related issues are addressed in [18].

– A composite service created from existing components may actually have more effects than specified by the service requester. This may not be desirable in certain situations since the client would observe extra (or even

⁷Service Types once defined and accepted would change only if major revisions are required. This is very much similar to evolution of classes in Java libraries which do not change often but might get deprecated or modified in subsequent releases.

unwanted) side effects unexpectedly.

4.3 Inheritance and polymorphism in SOC

As the services get deployed and composed in large scale enterprise systems, they would evolve to incorporate changing requirements. Replacing whole services would lead to breaking of contracts with existing clients since the behaviour might change. The solution is to add variants of the service that can coexist [16]. Interface inheritance can be effectively applied over here enabling different clients of same service to experience different behaviour.

More formally, given a base Service Type ST_{base} , the derived Service Type $ST_{derived}$ can be defined in terms of its stateDescription and profileTypes, as follows:

- $SD_{derived} \supset SD_{base}$, where $SD_i = \text{stateDescription}(ST_i)$, i.e. a derived service can add to the state inherited from the base service.
- $ST_{derived}$ may add new profileTypes to the set of profileTypes inherited from ST_{base} . It may also add new operationDescriptions to some or all of the profileTypes inherited from ST_{base} .
- For operations that are inherited from the base service, the IOPEs of each operationDescription $OD_{derived}$ are obtained as follows:
 - $I_{derived} = I_{base}$, i.e. the derived service has to be able to accept messages meant for the base service.
 - $O_{derived} \supset O_{base}$, i.e. the derived service can make more (but not fewer) outputs available than its base service, for a given operation.
 - $P_{derived} \vdash P_{base}$, i.e. preconditions of an operation in the derived service could be more general than (i.e. are entailed by) those of the corresponding operation in base service.
 - $(E_{derived} \not\vdash E_{base}) \wedge (E_{derived} \not\vdash E_{base})$, i.e. the effects of the base service do not necessarily entail the effects of the derived service and vice versa. In other words, they are independent. This is because the derived service may behave completely differently than the base service, though in most practical cases some of the effects of an operation in the base service would also occur in the corresponding operation of the derived service.

Apart from class-based inheritance, delegation has been used in prototypical languages [20] to provide inheritance at runtime. Delegation allows the parent of an object to be chosen at runtime instead of design time. Any messages not handled by the receiving object is forwarded to a parent thus selected. This approach can be used to introduce inheritance in the current SOAs where the notion of classes is missing. However we feel that delegation based approach is not suitable for SOC since the entites involved are business level entities typically engaged in cross enterprise integration. Enterprises would prefer to ensure that only those services are utilized that adhere to their specifications in terms of functionality provided, ownership and quality of service guarantees offered. Evaluating these preferences to search and select a parent at runtime, on a per-request basis would not be a scalable solution.

Discussion:

Inheritance in SOC provides following advantages:

- Inheritance and polymorphism can be used to provide differentiated service to different clients. In the helpline scenario, the telco might offer better service to customers of a preferred client by deploying better implementations for different components in the workflow. The exact implementation that serves a request would be selected at runtime based upon the request. This would require support from the runtime system, similar to WSIF but based on Service Type as the reference interface rather than WSDL.
- Enabling business process maintenance is another key benefit that inheritance and polymorphism can provide. New functionality, such as security etc. could be added into a derived Service Type whose instance could then replace the original one. This would enable both new and old clients to continue using the service.
- Interface inheritance can also be used to realize service composition at runtime.[11] illustrate that interface inheritance induces four forms of service composition.
- While Classification pushes binding from design time to deployment time, polymorphism enables pushing binding further to execution time.

Issues to be addressed are:

- Type checking mechanisms in OO systems prevented substituting a non-conforming object in place of an object of a particular type. Similar mechanism is needed for services to provide dynamic binding, enabled by inheritance and polymorphism, in a type-safe manner.

5 Conclusion

In this paper, we introduced a model for service oriented computing that is based upon the established and well tested object oriented paradigm. Moving beyond the debate surrounding dissimilarities between services and objects, we provided a possible direction for applying existing business integration experience for SOAs - a question raised by others [4]. We leveraged some of the existing Web Service technologies and generalized them to establish the OO concepts of Classification, Composition and Specialization as the basic architectural principles for SOC. Services in an SOA represent complete business functions at enterprise level (rather than individual programs or applications) [23], and the proposed model has an impact on what kind of features can be enabled for cross-enterprise services in SOAs. We also discussed benefits derived from the proposed concepts as well some of the issues that need to be resolved.

References

- [1] Open Grid Services Infrastructure (OGSI) V1.0. <http://forge.gridforum.org/projects/ggf-editor/document/draft-ogsi-service-1/en/1>.
- [2] WS Addressing, an XML serialization and standard SOAP binding for representing network wide "pointers" to services. <http://www.ibm.com/developerworks/webservices/library/ws-add/>.
- [3] V. Agarwal, K. Dasgupta, N. Karnik, A. Kumar, A. Kundu, S. Mittal, and B. Srivastava. A Service Creation Environment based on End to End Composition of Web Services. In *Proceedings of WWW*, May 2005.
- [4] S. Baker and S. Dobson. Comparing Service-Oriented and Distributed Object Architectures. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA)*, LNCS, Springer Verlag, 2005.
- [5] K. Birman, R. van Renesse, and W. Vogels. Adding High Availability and Autonomic Behavior to Web Services. In *26th International Conference on Software Engineering (ICSE)*, 2004.
- [6] G. Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Professional, 1993.
- [7] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. Web Services Architecture, W3C Working Group Note. <http://www.w3.org/TR/ws-arch/wsa.pdf>, Feb 2004.
- [8] J. Cardoso, A. Sheth, J. Miller, J. Arnold, and K. Kochut. Quality of Service for Workflows and Web Service Processes. *Journal of Web Semantics*, 1:281–308, 2004.

- [9] F. Curbera, W. Nagy, and S. Weerawarana. Web Services : Why and How. In *Workshop on Object Orientation and Web Services (OOWS), OOPSLA*, 2001.
- [10] V. D'Andrea and M. Aiello. Services and Objects: Open Issues. In *Proceedings of the First European Workshop on Object Orientation and Web Services (OOWS), Darmstadt, Germany*, July 2003.
- [11] V. D'Andrea, I. Fikoura, and M. Aiello. Interface inheritance for object-oriented composition based on model driven configuration . In *Proceedings of Second International Conference on Service Oriented Computing (ICSOC)*, Nov 2004.
- [12] M. J. Duftler, N. K. Mukhi, A. Slominski, and S. Weerawarana. Web services invocation framework (wsif). In *Workshop on Object Orientation and Web Services (OOWS), OOPSLA*, 2001.
- [13] I. Foster, J. Frey, S. Graham, S. Tuecke, K. Czajkowski, D. Ferguson, F. Leymann, M. Nally, I. Sedukhin, D. Snelling, T. Storey, W. Vambenepe, and S. Weerawarana. Modeling Stateful Resources with Web Services, Version 1.1. <http://www.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf>, March 2004.
- [14] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. Grid Services for Distributed System Integration. *IEEE Computer*, 35(6):37–46, June 2002.
- [15] R. Guerraoui and M. E. Fayad. OO Distributed Programming Is Not Distributed OO Programming. *Communications of the ACM*, 42(4), April 1999.
- [16] W. Haythorn. What is object-oriented design? *Journal of Object-Oriented Programming*, 7(1), Mar 1994.
- [17] A. Kumar, N. Karnik, and V. Agarwal. Usage Metering for Service Oriented Grid Computing. *International Journal of E-Business Research*, 2(1):78–105, Jan 2006.
- [18] A. Kumar, B. Srivastava, and S. Mittal. Information Modeling for End to End Composition of Semantic Web Services. In *Proceedings of the 4th International Semantic Web Conference (ISWC), Ireland*, Nov 2005.
- [19] D. Lea, S. Vinoski, and W. Vogels. Asynchronous Middleware and Services. *IEEE Internet Computing*, 10(1):14–17, Jan 2006.
- [20] H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *Proceedings of OOPSLA*, Oct 1986.
- [21] T. Modi. Splitting Hairs: Web Services Vs. Distributed Objects. *Web Services Pipeline*. <http://www.soa-pipeline.com/57704023>, Jan 2005.
- [22] OWL-S. <http://www.daml.org/services/owl-s/1.1/>, Nov 2004.
- [23] M. P. Papazoglou. Service-Oriented Computing: Concepts, Characteristics and Directions. In *4th International Conference on Web Information Systems Engineering (WISE)*, Dec 2003.
- [24] J. Pasley. How BPEL and SOA Are Changing Web Services Development. *IEEE Internet Computing*, 9(3):60–67, Mar 2005.
- [25] J. Rao and X. Su. A Survey of Automated Web Service Composition Methods. In *Proceedings of First International Workshop on Semantic Web Services and Web Process Composition*, July 2004.
- [26] E. Sirin and B. Parsia. Planning for Semantic Web Services. In *Semantic Web Services Workshop at 3rd ISWC*, 2004.
- [27] Simple Object Access Protocol. <http://www.w3.org/TR/SOAP/>.
- [28] A. Taivalsaari. On the Notion of Inheritance. *ACM Computing Surveys*, 28(3):438–480, Sept 1996.
- [29] W. Vogels. Web Services are not Distributed Objects: Common Misconceptions about the Fundamentals of Web Service Technology. *IEEE Internet Computing*, 7(6), 2003.
- [30] M. Wilson and B. Matthews. The Future of the World Wide Web? In *British National Conference on Databases (BNCOD)*, pages 4–15, 2004.
- [31] Web Services Description Language. <http://www.w3.org/TR/wsdl>.
- [32] The Web Services Resource Framework. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf, 2004.
- [33] L. Zeng, B. Benatallah, A. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-Aware Middleware for Web Services Composition. *IEEE Transactions on Software Engineering*, 30:311–327, 2004.
- [34] O. Zimmermann, P. Krogdahl, and C. Gee. Elements of Service-Oriented Analysis and Design: An interdisciplinary modeling approach for SOA project. <http://www.ibm.com/developerworks/webservices/library/ws-soad1/>, June 2004.