**Proc. IW-SOSE '06, Int'l Workshop on Service Oriented Software Engineering, Shanghai, China, May 2006 (to appear)**

# Separating Per-client and Pan-client Views in Service Specification

Colin Atkinson,
Dietmar Stoll
Chair of Software Technology
University of Mannheim
68161 Mannheim
+49 621 181 3914
{atkinson, stoll}@
informatik.uni-mannheim.de

Hilmar Acker,
Peter Dadam,
Markus Lauer
University of Ulm
89069 Ulm
+49 731 50 24131
{hilmar.acker, peter.dadam,
markus.lauer}@
uni-ulm.de

Manfred Reichert
University of Twente
7500 AE Enschede
+31 53 489 3705
m.u.reichert@cs.utwente.nl

## ABSTRACT
Service-oriented architecture is predicated on the availability of accurate and universally-understandable specifications of services which capture all the information that a potential user needs to know to use the service. However, WSDL, the most widely used service specification standard, only allows the syntactic signatures of the operations offered by a service to be described. This not only makes it difficult to specify context sensitive information, such as acceptable operation invocation sequences and drive service discovery through client-oriented requirements, it is also an inappropriate level of abstraction for a human friendly description of a service's capabilities. The current thinking is that context sensitive information such as operation sequencing rules should be described in an accompanying specification document written in an auxiliary language. For example, WS-CDL is a well known auxiliary language for writing choreography descriptions that capture interaction scenarios in terms of abstract roles and participants. However, this approach not only decouples the additional information from the core WSDL specification, it also describes it in terms of abstractions which may not match those used (implicitly or explicitly) by the service. In this paper we investigate this issue in greater depth, explore the different solution patterns and propose a new specification approach which rectifies the identified problems.

## Categories and Subject Descriptors
D.2.11 [Software Engineering]: Software Architectures: Patterns
D.2.1 [Software Engineering]: Requirements, Specification

## General Terms
Design, Theory

## Keywords
matching, service development, service specification



## 1. INTRODUCTION
Service-oriented architecture is predicated on the availability of accurate and universally-understandable specifications of services. Specifications not only carry the information needed to use services, they are also the means by which service users are introduced to suitable service providers. Finding the optimal form of service specification is therefore central to effective service-oriented development and the creation of efficient service-oriented architectures.

Since electronic services such as web services are usually accessed via asynchronous messaging or "remote procedure call" (RPC) style interaction mechanisms, the first generation of service specification standards such as WSDL [15][16] inevitably focused on the description of the "procedures" that a service offers. A WSDL service specification is essentially a description of the signatures of the procedures offered by the service including the types of their parameters and return values.

This provides the basic information that a service user needs in order to invoke the procedures of the service via a communication protocol such as SOAP [14]. As has been widely documented, however, there is a lot of additional information that a service user really needs in order to effectively interact with a dynamically discovered service. Chief examples include information about the order in which the operations of a service should be called (often called the protocol) and information about the semantics (i.e. the effects) of the operations. There is also a whole array of additional "meta" information that can be added to service specifications related to such issues as security, authentication and other concepts involved in the establishment of trust between service users and providers.

Not surprisingly, many research and standard-development initiatives have attempted to address these problems over the last few years. To express message sequencing information, or choreography as it is often called in the web service community, WS-CDL [13] (based on the Pi-Calculus [10][11]) has recently emerged as the industry-leading standard. For defining the semantics of services, OWL-S [12] is the leading language. In addition, there is a whole host of others languages that allow WSDL specifications to be annotated with additional "meta" data information to describe properties related to such things as

security, transactions, exception handling and quality of service etc. [8][7][6][5][4].

An important common characteristic of all these approaches, including those focusing on sequencing (choreography) and semantics, is that they do not change the underlying WSDL service specification - they retain the core specification in its original form and add additional information on top of it. In other words, they utilize the same basic operation signatures as WSDL, but add additional specification mechanisms to support the description of choreography and semantics etc. While this principle may be attractive from a compatibility point of view, however, it does not necessarily lead to specifications which convey the required information in the most efficient way.

The basic problem is that much of the information that needs to be added to capture such things as choreography and semantics is context specific, whereas the underlying WSDL specification is context independent. A WSDL specification of a service does not explicitly take into account the different roles or states of the users that may interact with it over time. These are only implicitly accommodated as operation parameters which carry role or state information in the form of IDs (e.g. session ID). However, since choreography, semantics, and the other higher-level "meta" information is usually context (i.e. identity) sensitive, this requires the additional information to be described in terms of these identity parameters.

Although it is clearly necessary for a service user to be able to derive the low-level procedural interface to a service in a mechanical way, it is not essential for this interface to be explicitly expressed in a service specification. On the contrary, as its name implies, a specification should be described at the level of abstraction that conveys the necessary information in the most effective way. Rather than base all specification information around the low-level procedural interface, therefore, it may be advantageous to specify a service using different abstractions which accommodate a more precise and concise description of context sensitive information. This principle is also in line with the basic tenet of model-driven development which calls for implementation-level, platform-specific descriptions of software artifacts to be derived automatically from more abstract and user friendly platform-independent specifications. As long as there is a well defined mapping from the specification to (one of) the implementation(s), a specification can take any form.

In this paper we present a strategy for specifying web services that simplifies the description of context sensitive information such as choreography and increases the level of semantic content. The basic idea is to raise the level of abstraction of a service specification and to make the implicit, context sensitive properties explicit. We developed this approach as part of the AristaFlow project [1] in order to simplify the problem of checking whether a component is suitable for plugging into a work flow process. However, it is not restricted to this area. We believe it provides an enhanced form of service specification for all purposes.

The rest of the paper is structured as follows. In section 2 we give an overview of the problem and in sections 3 and 4 we present the patterns and concepts that can be used to overcome them. We finish by presenting a better way to specify services and discuss how to map these specifications to platform specific descriptions.

## 2. THE PROBLEM

To understand the limitations of WSDL style service specifications, consider the following web service "interface" which is fairly typical of stateful services. This service provides support for a fairly common requirement in business applications, the creation and manipulation of orders. The operations of the service allow users to create order objects, add one or more items to order objects, and once payment details have been defined and checked, to "place" the orders. A user can place several orders during a given session, and can work on several orders simultaneously.

```
SessionedOrderManager {
  createSession return String
  authenticate (String userID, String password)
  createOrder return String
  defineOrderCustomer (String info, String orderID, String sessionID)
  defineOrderPayment (String info, String orderID, String sessionID)
  addOrderItem (String name, String orderID, String sessionID)
  deleteOrderItem (String name, String orderID, String sessionID)
  calculateOrderTotal (String orderID, String sessionID) return Euro
  checkOrderPayment (String orderID, String sessionID) return Boolean
  placeOrder (String orderID, String sessionID)
}
```

We have presented this interface in a Java like syntax to avoid the clutter of the XML element tags in a full WSDL specification. The intent is not to imply the use of any particular programming language or standard, but to list the signatures of the operations that the web service, as an object abstraction, offers to users. In a WSDL document, this would appear in the portType and message definition and would consist of the operation specifications. A complete WSDL document would of course also have service and binding parts, but this is not of interest here.

From the point of view of the service as a whole the execution of the methods is not governed by any particular sequencing rules. Because the service is designed to support multiple concurrent users, the operations are arbitrarily interleavable and there is therefore no protocol or set of sequencing rules. To use operating system terminology, the operations are multiply "reentrant". From the point of view of an individual user, however, the invocation of operations is very definitely governed by a set of sequencing rules (or a protocol) since the operations applied to a given order must be applied in a meaningful sequence.

More specifically, to successfully create and place an order a customer must perform the following tasks (by invoking the corresponding operations) in the following order -

- create a new order
- addItem (aI) or removeItem (rI) multiple times, with the number of addItems invocations always having been greater than the number of removeItems
- defineCustomer (dC) and definePayment (dP), in any order
- calculateTotal (cT)
- checkPayment (cP)
- place (p)

When the order abstraction is considered alone, this sequencing information can be specified or modeled in a straightforward way. For example, using a regular expression language [9] the above interleaving of operations can be easily expressed as follows.

*( addItem | deleteItem )*$^{+}$ .
*((defineCustomer . definePayment) |*
*(definePayment . defineCustomer))* .
*calculateTotal . checkPayment . place*

**Figure 1. Protocol as Regular Expression**

It can also easily be expressed in the form of a state transition diagram such as in figure 2.

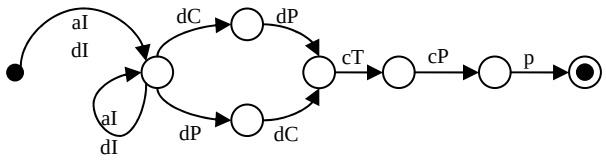


**Figure 2. Order Protocol Diagram**

These specifications are relatively concise and simple because in both cases all operations appearing in the specification apply to a single object. The identity of the object is thus implicit. However, at the level of the OrderManger service as a whole the issue of identity is important, because the ordering requirements only make sense in the context of individual objects. Therefore, to specify the above ordering constraints (on individual order objects) in terms of the operations exported by the OrderManager service (which operate on multiple orders), the order identifier must explicitly be taken into account. In the above version of OrderManager this is the role of the orderID parameter of the order operations.

The same also holds for the specifications of the effects of the operations in terms of pre and post conditions. In the simple, context sensitive case, the pre and post conditions of all the operations would by default apply to the same instance of an object. In the service-wide, context independent case, however, the identity of the order object must always be included in the specification.

A good example of how the specifications of protocols are complicated by the need to take object identity into account is given by the WS-CDL specification of the above sequencing rules. This is shown below in figure 3.

```
<package xmi:version="2.0" xmlns:om="http://www.example.com/order
  name="orderCDL" version="1.0" targetNamespace="…" …>
…
<informationType name="StringType" typeName="xsd:string"/>
<informationType name="EuroType" typeName="om:Euro"/>
<informationType name="addItemMessageType"
   typeName="om:addItemMessageType"/>
<informationTypes name="removeItemMessageType"
   typeName="om:removeItemMessageType"/>
<informationType name="paymentInfoType"
   typeName="om:paymentInfoType"/>
<informationType name="customerInfoType"
   typeName="om:customerInfoType"/>
<informationType name="paymentOkInfoType"
   typeName="om:paymentOkInfoType"/>
<informationType name="TotalType" typeName="om:TotalType"/>
...

<token name="orderID" informationType="StringType">

<tokenLocator token="orderID" informationType=
  "addItemMessageType" query="/RI/orderID"/>
<tokenLocator token="orderID" informationType=
  "removeItemMessageType" query="/RI/orderID"/>
<tokenLocator token="orderID" informationType=
  "paymentInfoType" query="/RI/orderID"/>
<tokenLocator token="orderID" informationType=
  "customerInfoType" query="/RI/orderID"/>
<tokenLocator token="orderID" informationType=
  "paymentOkInfoType" query="/RI/orderID"/>
<tokenLocator token="orderID" informationType=
  "TotalType" query="/RI/orderID"/>

<channelType name="customerOrderChannelType"
   referenceToken="customerRef" >
      <identities description="orderID" tokens="orderID"/>
</channelType>

<choreography name="orderChoreo" root="true">
...
<activitiy type=" Interaction" name="addItemActivity"
  operation="addItem" channelVariable="customerOrderChannelType"
   ...>
   <exchangeDetail name="addItem" type="addItemMessageType"
      action="Request"/>
</activity>
```

**Figure 3. WS-CDL description**

In WS-CDL, the protocol between a user and a service is defined primarily in terms of roleTypes and channels. At each place where an operation of the service is invoked, a message (which has an informationType) is sent through a channel. A channel represents a connection between one client and one service provider. One common way to create a WS-CDL description is to attach a token (here: the orderID) to the channel, so that the provider can uniquely identify the order as it receives a message. Nevertheless, for each message exchanged, a tokenLocator must be defined to identify the part of the message which holds the orderID.

Because WS-CDL and WSDL are written against the whole interface, in each case the orderID must be included in the message. When the processing of order objects is serialized (i.e. one order (object) has to be created and placed before the next is created), this need to refer to orderIDs is merely inconvenient and cumbersome. However, if processing of orders can themselves be interleaved (i.e. multiple orders of one client can be in various stages of processing at the same time), then the need to explicitly refer to IDs can become a serious problem. This situation cannot be cleanly handled in WS-CDL because one has to explicitly fix the number of order processes that can be underway at any given time, given one instance of a server. The idea, of course, is that this number should be dynamically changeable.

Including the orderID parameter in the procedural interface of the OrderManager service (for the operations that manipulate Order objects) is unavoidable, because the whole point of this service is to take responsibility for managing order objects on behalf of users. Thus, at the (SOAP) method invocation level, an orderID parameter must clearly be included. This does not mean, however,

that order identifier parameters need to be included in the specification of services. As long as clear patterns are applied, and a systematic convention is used to handle object identity in the complete interface, it should be possible to fully specify the service without explicitly elaborating the details of object identification. By "fully specifying", we mean that the service can be specified in such a way that the full, implementation-level interface can be derived unambiguously in a simple and straightforward way.

The big advantage of making object identification implicit, and specifying the service in a context sensitive way, is that the specification of context sensitive information such as operation sequencing rules or pre and post conditions is greatly simplified. In the next two sections we introduce the two patterns which we believe assist in the attainment of these benefits.

# 3. THE MANAGER PATTERN

Most stateful services, such as the OrderManager service from the previous example, have the role of managing instances of some other abstraction. In this case the OrderManager is responsible for "managing" instances of a simple abstract data type (ADT) or class: Order. By "managing" we mean that the service allows instances of the managed ADT to be created and destroyed and operations of the ADT to be applied to identified instances.

Since this relationship is so common in stateful services we characterize it as a pattern – the manager pattern. Figure 4 illustrates the structure of this pattern for the Order – OrderManager example. On the left hand side we have the core abstraction pictured as a UML class with all its methods. This is a simple Abstract Data Type (ADT). On the right hand side we have the "manager" object which is derived from it. The purpose of the "manager" object is to support the creation and deletion of the basic ADT objects and to allow each of the ADT operations to be applied to each instance of the ADT identified by an identifier.

Apart from the addition of the creation and deletion operations, therefore, the main difference between the core ADT and its manager is the addition of the extra "ID" parameter to the methods to identify which instance is intended. Notice also, however, that the name of each of the operation has also subtly changed. The name of the addItem method in the managed ADT has changed to orderAddItem in the manager to reflect the fact that its operations do not affect its internal state directly but that they add items to the Order objects. As long as these name changes are performed systematically the manager abstraction can be derived from the base ADT simply and unambiguously. Stated differently, if a service user knows the operation signatures of the managed ADT, it also knows the operation signatures of one of its manager objects, as long as it is derived systematically from it.

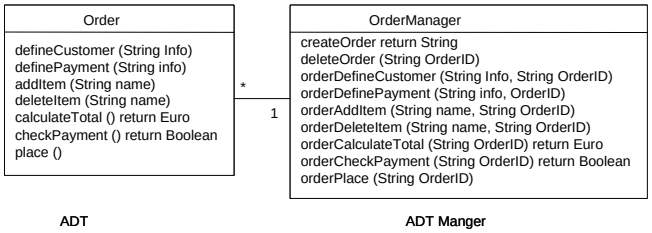


**Figure 4. Instance of the Manager Pattern**

The Manager abstraction bears some resemblances to other well known patterns and software engineering concepts. For example, it subsumes the factory pattern since it provides methods for creating and deleting objects of a specific type.

Figure 4 shows a concrete application of the manager pattern in the context of the Order example. In figure 5 below we show the generalized structure of the pattern.

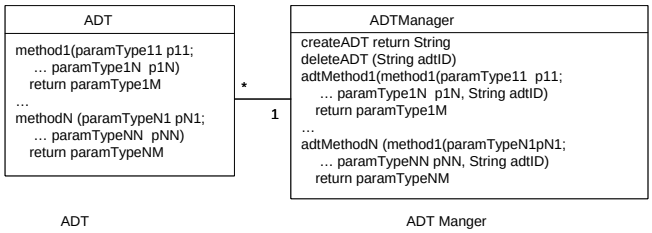


**Figure 5. Manger Pattern Structure**

One of the key principles of our service specification approach is that the abstractions managed by the service should be described explicitly and independently. Thus, the specification for the OrderManager web service would contain a description of the managed ADT – Order, as well as (or possibly even instead of) a description of the manager. This is illustrated below using the same syntax as that in figure 1.

Order {

defineCustomer (String info)
definePayment (String info)
addItem (String name)
deleteItem (String name)
calculateTotal () return Euro
checkPayment () return Boolean
place ()

*( addItem | deleteItem )*$^{+}$ .
*((defineCustomer . definePayment) |*
*(definePayment . defineCustomer))* .
*calculateTotal . checkPayment . place*
}

The big advantage of separating out the specification of the managed ADT in this way is that it can be accompanied by the abstraction-specific sequencing constraints. These can take a straightforward form because they can be based on the assumption that all methods in a sequence operate on the same ADT instance. The example specification above includes the simple regular expression sequencing specification from above. Pre and post conditions could also be added in the same context-specific way.

## Composition of Managed Objects

An abstraction that plays a particularly important role in client-server architectures such as service-oriented architectures is that of a session. A session is the abstraction which is generally used to store the state of a conversation between users of a service and providers of a service, when, as is generally the case in service-oriented architectures, there can be multiple users of a single service. The session abstraction, or more specifically the session identifier, is used to provide service users with the illusion that they are the sole user of the service.

In its simplest form, Session is a very simple ADT. The only functionality which it usually offers is a procedure to authenticate

users, usually via the checking of user names and passwords. Figure 6, below shows the manager pattern applied to the session ADT.

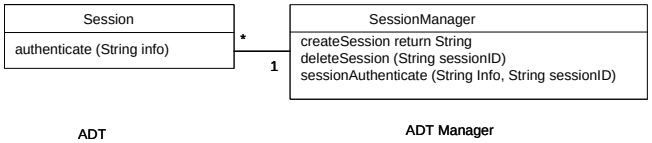


**Figure 6. Instance of the Session Pattern**

Of course, a service which simply offers the capability to create sessions and nothing else would be of little use. Useful services always combine session management with other functionality, such as for example, the management of additional abstractions. Therefore, it is necessary to have a way of combining managed ADTs within the context of a single manger service. Combining session management with other management services is only one example, albeit a very important one.

The composition of session management with the management of other ADTs cannot be done in a naive way, however. For example, in the case of a session-based order management service it would not make sense to simply aggregate all the methods from the OrderManager abstraction and the SessionManager abstractions into a single service as illustrated below. The problem is that session management should not take place independently of order management, it should be "superimposed" on top of it. The version of the SessionOrder manager below is unrealistic because the management of orders (i.e. the invocation of order operations) takes place outside the context of a session. The whole point of adding session management to the service, however, is that all functionality offered by the service should be performed within the context of sessions.

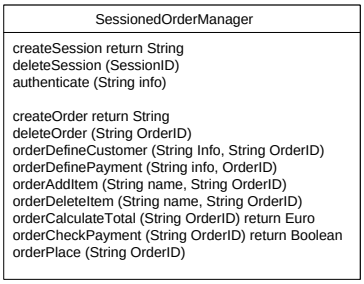


**Figure 7. Naive Session Manager**

The version of the session manager in figure 8a) provides a much more useful combination of the session and order management abstractions because it superimposes the session management service "on top of" the order management service. This is evidenced by the fact that all of the order management operations have an additional sessionID parameter, as well as an orderID parameter, to allow the session to be identified.

The "superimpose on top of" relationship is of course not very well defined here. In general, when fully fleshed out, this approach to service specification will include several standard combination operators such as for example "union", which give the naive case in figure 7, and "superimpose" which gives the realistic case in figure 8a). Although the "union" operator is unrealistic in this particular example, it can be useful in other scenarios.

# 4. PAN-CLIENT AND PER-CLIENT INTERFACES

Capturing the session management characteristics of a service by superimposing the session manager abstraction on top of the other services is a reasonable approach. However, session management superimposition is such a universal requirement in service-oriented architectures that it makes sense to treat it as a special case. In other words, we believe it is valuable to define special names for the "session managed" and "session unmanaged" view of a service. This represents the second key idea in our approach to service specification.

With this observation in mind, we believe that all session managed services can have their interfaces represented in two basic forms – the pan-client and the per-client interfaces. The motivation for these names is simple. Since the whole purpose of session management is to give clients the illusion that they are the sole user of a service, a representation of the service interface in which session identification is implicit corresponds to a client's "private" view of the service. The name "per-client" interface is thus intended to capture this form. On the other hand, a representation of the service interface in which the session ID is made explicit (and thus all methods have Session ID parameters) corresponds to a global view of the service in which the existence of multiple concurrent clients is made clear. The name "pan-client" interface is thus intended to capture this form. The relationship between these two representations of the SessionedOrderManager service is shown below in figure 8.

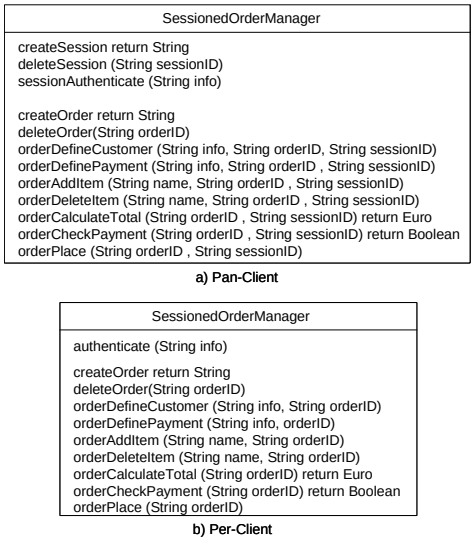


**Figure 8. Pan and Per-Client Interface**

As can be seen from this figure, the per-client view of the service is much simpler and more understandable than the pan-client view. The latter is basically the full, WSDL-style specification of the service in which the complete invocation signature of each procedure is fully elaborated. The per-client view also provides a complete picture of all the procedures offered by the service. However, it does so without explicitly describing the session identification information. In effect, the per-client view resembles the interface that clients of the service would have if they each genuinely had a dedicated service provider which was exclusively servicing their own needs and maintaining their own private interaction state. This view much more closely matches the view

that human users of a service-based application receive of a service – namely, the view that they are the exclusive user. Browsers and other client applications are deliberately designed to "hide" session management issues from the human user.

As with the manager pattern described in the previous section, introducing the separate concepts of the per and pan-client views of a service interface only makes sense if there is a clear relationship between them, such that one can be derived from the other simply and unambiguously.

### Enhanced Specification of Services

To illustrate how the concepts explained above can be used to enhance the specification of services, in figure 9 we depict all the different abstractions and views that can be created for the SessionedOrderManager service discussed above. To highlight the variety of abstractions and views, we introduce an additional customer survey abstraction as a managed ADT.

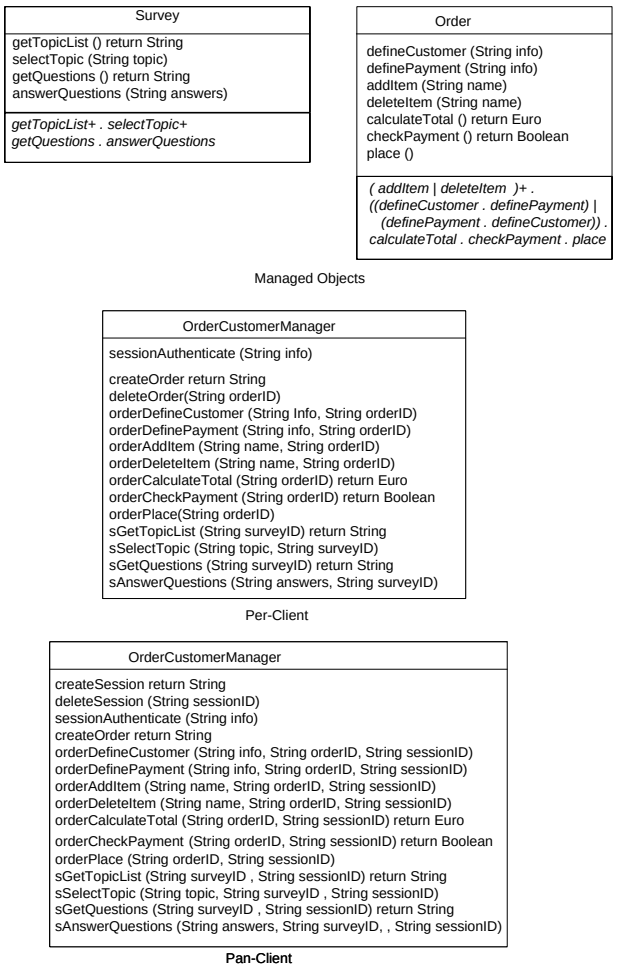


**Figure 9. Different Abstractions and Views**

It is not the intention that all the different parts should be explicitly documented within a service specification. On the contrary, the idea is that a service specification would contain only the minimum information necessary to enable all these views to be generated. Since derivability is generally from top to bottom, i.e. the pan-client view can be derived from the per-client view and the per-client view can be derived (partially) from the managed ADTs), this implies that the bulk of the information within a specification will be in the form of the managed objects and the per-client interface.

When the different abstractions are separated out in this way, the relevant sequencing constraints can be added at the most relevant place as shown in figure 9. Sequencing constraints on the operations of the managed abstraction can be defined as part of their specification, and any additional per-client constraints at the level of the whole service can be added to the per client version. Similarly any additional constraints that span all clients can be added at the pan client level.

If a suitable language were available to describe how the managed ADTs on top in figure 9 are composed into the combined service (i.e. via union or superimposition operators) the per-client interface need not be represented in the explicit form shown either. Instead, I could be described using a combination expression together with any additional sequencing rules that define the order in which instances of the different abstractions are managed. Analogous operators can be envisaged to describe how the pan-client interface is derived from the per-client interface in unusual cases.

## 5. CONCLUSION

In this paper we have highlighted some shortcomings in the current techniques and languages used to specify services in the context of service-oriented architectures, in particular with regard to the description of context sensitive information such as protocols and have put forward some ideas for addressing them. These revolve around two specific concepts – the concept of the manager pattern in which the objects "managed" by a service are explicitly specified and the concept of "per-client" interfaces in which the session information is implicit. The basic idea is to specify services in terms of these concepts, wherever possible, rather than in terms of the usual pan-client interfaces captured in WSDL documents.

For two of the three main roles of service specifications, this brings significant benefits. For service usage, where a service client interacts with a service provider via messaging services such as SOAP, the approach brings no real advantages. However, it has no disadvantages either, since a key tenet of the approach is that the pan-client interaction must be unambiguously derivable from the information in a service specification, regardless of the form that it takes. As explained in the paper, provided the patterns are applied systematically, pan-client interfaces can always be derived from the managed ADTs and the per-client interface.

For service discovery, the approach allows the matching of service providers to service users to be driven from a context sensitive (e.g. client oriented) viewpoint that can include such things as protocol descriptions and operation semantics (pre and post conditions). At present, service discovery algorithms usually only take the pan-client procedure signatures into account when searching for components that match a particular service requirement. However, the ability to satisfy context sensitive requirements is a fundamental aspect of a contract between a service user and provider, and thus should be taken into account when searching for components. Moreover, the additional information needed for describing context sensitive information (e.g. session IDs) should be derived automatically from a user friendly description. Instead of having to specify the whole pan-

client view for service discovery, only a specification in terms of the ADTs has to be provided.

Having a way of searching for services with specifications that match the user's perspective and considering protocol information when plugging services into workflow processes was the original motivation for this research in the context of the AristaFlow project.

For service development, where a software engineer actually creates the service specification, the approach also has significant advantages. As can be seen from figure 9, even without protocol information, the managed ADTs and the per-client interface are much simpler and more concise than the pan-client representation of a component. When protocol (method sequencing) information is taken into account the difference in clarity and expressiveness is even greater. Human developers will have a much easier time defining services, and the associated sequencing rules, in terms of managed ADTs and per-client interfaces than in terms of the traditional pan-client interfaces in WSDL.

Since it reinforces the idea of defining services in terms of different viewpoints, the approach works particularly well with view-based methods for describing or modeling components, such as the KobrA approach [2]. The explicit modeling of all abstractions handled by a component is already a key aspect of this method, and is strongly reinforced by the approach described in this paper.

The idea of defining distinct viewpoints on a service has some things in common with the role object pattern of Dirk Bäumer et al [3]. The key difference is that the goal of those patterns is to allow a given service to take on different roles during its lifetime, based on the needs and desires of specific clients. These roles may be defined and attached to the service dynamically. This is not the goal of our approach however. In our case the different views on the service are fixed at definition time.

The other important point to note about the approach is that it enhances the level of semantic information in service specifications. It does this by clearly denoting the role and "meaning" of the additional ID parameters that are introduced in each transformation step. The fact that the derivation of representation abstraction can only take place from top to bottom in figure 9 is significant because it highlights the fact that information is lost as the context sensitive management and session information on the top is "folded into" the pan-client view in the lower part of the figure. In other words, fully elaborating the managed ADTs and the per-session interface provides additional semantic information about the ID parameters which is simply not present in the WSDL-style pan-client view.

## 6. ACKNOWLEDGEMENTS

This research is being performed as part of the AristaFlow Project which is supported by the State of Baden-Württemberg.

## 7. REFERENCES

[1] AristaFlow Project, Next Generation Enterprise Process Management: Component-oriented Development of Adaptive Process-oriented Enterprise Software, http://www.aristaflow.de/, visited Feb. 2006

[2] Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wüst, J., Zettel, J.: *Component-based Product Line Engineering with UML*, Addison Wesley, 2002.

[3] Bäumer, D., Riehle, D., Siberski, W., and Wulf M., *The Role Object Pattern*, The 4th Pattern Languages of Programming Conference, Sept. 3-5, 1997

[4] BEA, IBM, Microsoft, SAP; *Web Services Policy Framework (WS-Policy)*, http://www-106.ibm.com/developerworks/library/ws-polfram/, May 2003, visited Feb. 2006

[5] IBM, BEA Systems, Microsoft, Arjuna, Hitachi, IONA. *Web Services Transactions specifications*, August 2005, visited Feb. 2006

[6] IBM, Microsoft, BEA, RSA Security, VeriSign *Web Services Federation Language (WS-Federation)*, ftp://www6.software.ibm.com/software/developer/library/ws-fed.pdf, July 2003, visited Feb. 2006

[7] IBM, Microsoft, RSA Security, VeriSign, et al. Web Services Secure Conversation Language (WS-SecureConversation), *ftp://www6.software.ibm.com/software/developer/library/ws-secureconversation.pdf*, May 2004, visited Feb. 2006

[8] IBM, Microsoft, VeriSign. *Web Services Security (WS-Security) 1.0*, http://www-128.ibm.com/developerworks/webservices/library/ws-secure/, April 2002, visited Feb. 2006

[9] Malan, R., Letsinger, R., and Coleman D., Editors, *Object-Oriented Development at Work: Fusion in the Real World*, Prentice Hall/HP Press, 1996.

[10] Milner, R. *Communicating and Mobile Systems: the Pi-Calculus*, Cambridge University Press, May 1999

[11] Milner, R., Parrow, J., and Walker, D. *A calculus of mobile processes*, part I/II, Journal of Information and Computation, 100:1–77, September 1992

[12] The OWL Services Coalition, *OWL-S 1.0: Semantic Markup For Web Services*, Whitepaper, http://www.daml.org/services/owl-s/1.0/, Jan. 2004, visited Feb. 2006

[13] W3C, *Web Services Choreography Description Language Version 1.0*, http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/, Nov. 2005, visited Feb. 2006

[14] W3C. *Simple Object Access Protocol (SOAP) 1.2*, http://www.w3.org/TR/2003/REC-soap12-part1-20030624/, visited Feb. 2006

[15] W3C. *Web Services Description Language (WSDL) 1.1*, March 2001, http://www.w3.org/TR/2001/NOTE-wsdl-20010315, visited Feb. 2006

[16] W3C. Web Services Description Working Group, WSDL 2.0, http://www.w3.org/2002/ws/desc/, visited Feb. 2006