# Semantic-based Mashup of Composite Applications

Anne H.H. Ngu, Michael P. Carlson, Quan Z. Sheng, *Member, IEEE,* and Hye-young Paik

**Abstract**—The need for integration of all types of client and server applications that were not initially designed to interoperate is gaining popularity. One of the reasons for this popularity is the capability to quickly reconfigure a composite application for a task at hand, both by changing the set of components and the way they are interconnected. Service Oriented Architecture (SOA) has recently become a popular platform in the IT industry for building such composite applications with the integrated components being provided as Web services. A key limitation of solely Web service based integration is that it requires extra programming efforts when integrating non Web service components, which is not cost-effective. Moreover, with the emergence of new standards, such as Open Service Gateway Initiative (OSGi), the components used in composite applications have grown to include more than just Web services. Our work enables progressive composition of non Web service based components such as portlets, web applications, native widgets, legacy systems, and Java Beans. Further, we proposed a novel application of semantic annotation together with the standard semantic web matching algorithm for finding sets of functionally equivalent components out of a large set of available non Web service based components. Once such a set is identified the user can drag and drop the most suitable component into an Eclipse based composition canvas. After a set of components has been selected in such a way, they can be connected by data-flow arcs, thus forming an integrated, composite application without any low level programming and integration efforts. We implemented and conducted extensive experimental study on the above progressive composition framework on IBM's Lotus Expeditor, an extension of a SOA platform called the Eclipse Rich Client Platform (RCP) that complies with the OSGi standard.

**Index Terms**—Mashup, composite applications, Web services, semantic Web, semantic annotation.

## 1 Introduction

Composite applications are a line of business applications constructed by connecting, or wiring, disparate software components into combinations that provide a new level of function to an end user without the requirement to write any new code. The components that are used to build a composite application are generally built within a Service Oriented Architecture (SOA). Many of the first SOA platforms exclusively relied on Web services (WSDL-based) as components in the composite application. The composition is done generally using process based languages such as BPEL [1]. The Web-service-only integration framework requires extra programming efforts when integrating with non-Web service components, which is not cost effective. With the emergence of new standards, such as Open Service Gateway Initiative (OSGi) [2], the components used in composite applications have grown to include more than just Web services. Components can be built from web applications, portlets, native widgets, legacy systems, and Java Beans. There are several challenges to mashing up non-Web service components into composite applications, especially when components are developed at different times, by different groups, using different technologies, naming conventions, and structures.

Firstly, a given enterprise may have hundreds of similar components available for mashup in a catalog, but manually searching and finding compatible and complementary components could be a tedious and time consuming task. For example, in a portal environment, it is possible to query the system for the available components. However, the list returned is typically based on criteria that have no relevance to the application assembler (e.g., in alphabetical or last update time). Interfaces like Google Code Search [3] allow the developers to search application code, but it does not allow them to search using the higher level concepts of a component or a model. On the other end of the spectrum, having to manually classify and describe every aspect of components for browsing and searching can be a painstaking task when handling a large number of components.

Secondly, none of the existing OSGi environments provides a way to leverage the semantic search techniques that have been developed to assist users in locating compatible components like in Web service-based composite applications. Unlike Web services, many non-Web service components have graphical user interfaces built from technologies such as portlets, Eclipse Views, and native application windows. Moreover, there is currently no standard way of categorizing and cataloging components for use in a composite application. Rather, components are discovered by assemblers who must hunt around the Web, in documentation, and searching the locally installed system. This does not provide an easy and manageable means of finding and selecting components. Depending on the technology used or the type of user interface being presented, certain components may not be valid for use in a particular composite application. Discerning this could be a tedious process up front, or could result in repeated cycles of trial and error, especially when the target environment supports a variety of technologies.

- *Anne H.H. Ngu is with the Department of Computer Science, Texas State University, TX 78666-4616, USA. E-mail: angu@txstate.edu.*
- *Michael P. Carlson is with IBM, Quan Z. Sheng is with the University of Adelaide, Hye-young Paik is with the University of New South Wales.*

After suitable components have been discovered, the assembly of composite applications should not require tedious and detailed programming as required of a typical software developer. Users, at least the savvier users, should be able to compose applications with minimal training. For a call center in an enterprise, this may mean being able to assemble a composite application on the fly. This may involve, for instance, extracting a piece of caller's information from one application and feeding it as input to other applications that are currently running on her desktop for other contextual information that might help in answering pressing queries. For a user in a small business, this may mean assembling a GPS routing application together with the list of errands or deliveries for the day and producing a more optimized route.

We have developed a novel approach that enables progressive composition of non Web service based components such as portlets, web applications, native widgets, legacy systems, and Java Beans. Our approach exploits semantic annotation together with the standard semantic web matching algorithm for finding sets of functionally equivalent components out of a large set of available non Web service based components. The identified components can be connected by data-flow arcs in an Eclipse based composition canvas, thereby forming an integrated, composite application without any low level programming and integration efforts. The main contributions of this paper are as follows:

- The paper first shows that existing techniques, technologies, and algorithms used for finding and matching Web service components (WSDL-based) can be reused, with only minor changes, for the purpose of finding compatible and complementary non-Web service based components for composite applications. These components may include graphical user interfaces, which are not artifacts described in Web service components. By building on the techniques initially developed for Web services matching, finding useful and valid components for composite applications using high level concepts is possible. This enables the progressive construction of composite applications by end users from a catalog of available components without deep knowledge of the components in the catalog. This is an advantage over existing mashup tools which require mastering of varying degrees of programming skills [4], [5].
- Being able to automatically find components suitable for a composite application is critical for any mashup toolkits. We demonstrate in this paper how the additional characteristics of components, specifically graphical user interface details, can be modeled, described using Semantic Annotation for Web Service Description Language (SAWSDL) [6] and matched in a similar fashion to the programmatic inputs and outputs of Web service-based components. Though similar in some respects to Web service-based components, our experimental study shows that these additional characteristics of a component allow for further match processing logic to be used to provide better results when searching for components.
- This paper shows, through sample applications, that by considering the unique characteristics of a component (i.e., coexistence of graphical user interface description), and new techniques of merging semantic descriptions across multiple components, a much more accurate search result for compatible components can be achieved.

The paper is organized as follows. Section 2 outlines the overall architecture of our progressive composition framework. Section 3 details the concepts of composite application matching, merging multiple components into a single descriptive format for matching, and modeling of component's GUI characteristics. Section 4 provides a set of experiments, results, and analysis of progressive composition framework based on semantic web matching technique with SAWSDL annotations. Section 5 describes the related work and Section 6 provides the conclusion and future work.

## 2 Progressive Composite Application Framework

### 2.1 Application Components

The application components referred to in this paper generally contain GUI interfaces, built from technologies such as JFace, JSPs, Servlets/HTML, Eclipse Standard Widget Toolkit (SWT), Swing, native windowing systems, etc. Like Web services and Enterprise Java Beans, these application components can take programmatic inputs and produce programmatic outputs. The programmatic inputs will generally cause changes in the graphical user interface, and user interaction with the graphical user interface will cause the programmatic outputs to be fired. An example of an application component referred to in this paper is a Portlet [7].

### 2.2 Lotus Expeditor Composite Application Framework

We adopt the IBM Lotus Expeditor [8] platform to develop application components and to mashup composite applications. Lotus Expeditor is an extension of an Eclipse Rich Client Platform that complies with OSGi standard and SOA architecture. The Expeditor contains a Composite Application Infrastructure (CAI) and an associated Composite Application Editor (CAE). Figure 1 is a simplified architecture diagram of Lotus Expeditor Framework. CAI is the runtime environment for the composite applications. It has two main components called Topology Manager and PropertyBroker. The Topology Manager is responsible for reading and interpreting the layout information stored in the composite application. The PropertyBroker is responsible for passing messages between independent components within CAI, in other words, it performs data mediation for composite applications. The CAE editor is used to assemble, and wire components into composite applications without the need for the underlying components to be aware of each other at development time and without the user having to write any additional codes. The desired components can simply be dragged and dropped to add them to a composite application. The adding, removing, and wiring can be done in an iterative/progressive fashion to allow the assembler to refine the composite application. This declarative data-flow

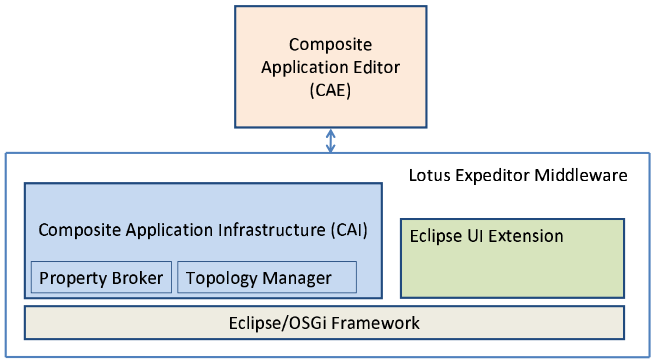

Composite Application Editor (CAE)
Lotus Expeditor Middleware
Composite Application Infrastructure (CAI)
Property Broker
Topology Manager
Eclipse UI Extension
Eclipse/OSGi Framework

Fig. 1. Lotus Expeditor Composite Application Framework

like wiring of components is one of the main advantages of Lotus Expeditor. The wired components can be saved in an XML file and written to local file system, hosted in a web server/portal server, or placed in Lotus Domino NSF database for reuse or customization by other users.

The programmatic inputs and outputs of an application component in CAI are described using WSDL. Typically, the associated WSDL files for CAI components are created as part of the component development process. In the current implementation of Lotus Expeditor, the WSDL files for application components in CAI do not include the graphical user interface type (e.g. JSP, SWT, AWT, Swing, etc.). The composite application assembler must have previous knowledge of component interfaces they are restricted in and the types of GUI technologies they can use. For example, if the composite application deployment platform does not provide support for portlet interfaces, an assembler must know which components in the repository are built from portlets and specifically avoid those when assembling the composite application. Lotus Expeditor also does not provide a way for finding compatible and complementary components from a catalog of existing components based on *components' capabilities*. We extended Lotus Expeditor Workbench with a new menu item called *Analyze Composite App* that opens a dialog box for user to search for the desired components to use for composition based on high-level semantic concepts.

## 2.3 Sample Composite Applications

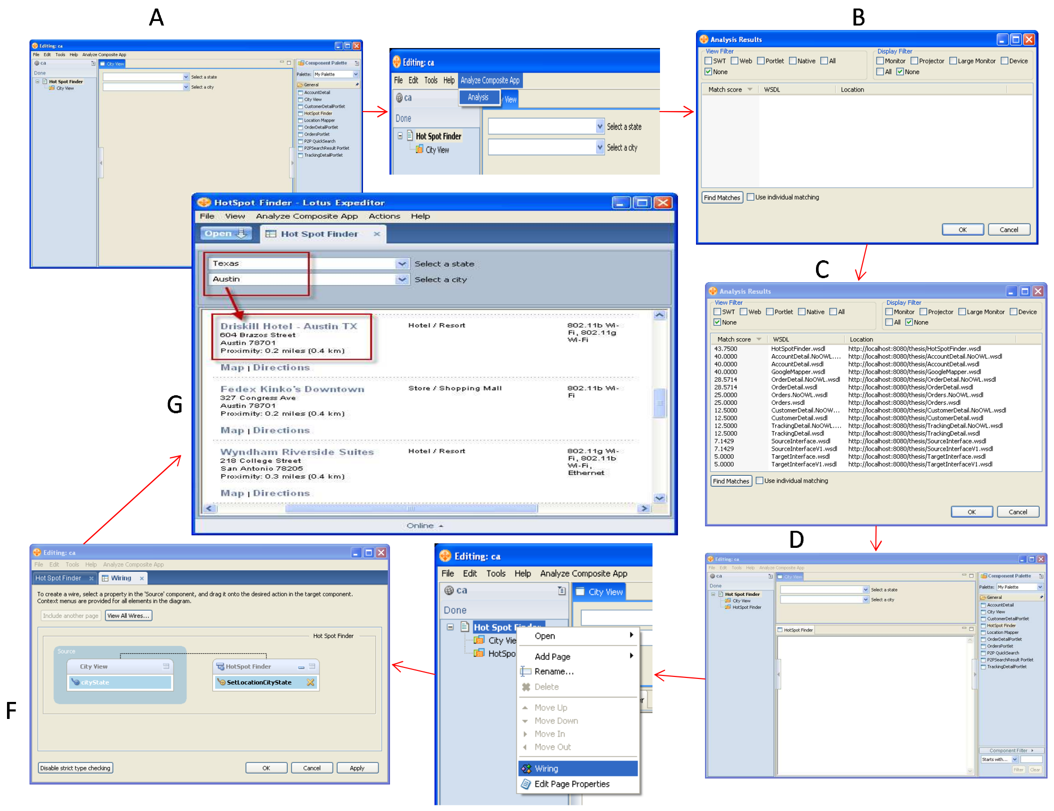
In this section, we describe the functionalities of two sample applications used in the experimental study of our mashup framework. These two sample applications were selected because they represent two different types of valid composite applications. The `HotSpotFinder` represents a hybrid composite application that includes Eclipse SWT GUI and a web application (jWire) accessed via a web browser (RESTful Web service). The `order tracking` application represents a real-world composite application that could be deployed to a cooperative portal environment like IBM WebSphere Portal or BEA WebLogic Portal. The `HotSpotFinder` composite application is composed of three separate components.

- `CityStatePicker` is implemented as an Eclipse UI Extension class, which allows a user to select a state and a city from two separate drop down lists. After both city and state have been selected, the component publishes the city and state as a single output.
- `HotSpotFinder` is implemented as an instance of the Eclipse SWT Browser, which is programmatically driven to different URLs based on the inputs. In order to provide interesting content, the JiWire [9] website is accessed by the browser. The `HotSpotFinder` takes as input a city and state. When this input is received, the browser is directed to a URL (constructed dynamically with city and state as input) on the JiWire website, which provides a list of wireless Internet access points in the given city. By double clicking on an address shown in the `HotSpotFinder`, the address is published as an output.
- `GoogleMapper` is implemented as an instance of the Eclipse SWT Browser, which takes as input an address, based on this address, the browser loads a map for the address using Google Maps to provide the actual content.

The above three components can be developed separately by different programmers using different technologies. In order for them to be made available for our composition framework, they must be imported into Lotus Expeditor. The WSDL file for each component must be created and annotated as shown later in Section 3.1.

The `OrderTracking` application is composed of five individual components, with several inputs and outputs. The individual components are all built as portlets. The base code for this scenario was taken from the Cooperative Portlets sample provided as part of Rational Application Developer 7.0. The same sample is described in detail in [7]. The code was reused with only minor changes; small errors were corrected in the application code. The five components are:

- `Orders Portlet` displays a list of existing orders for a specific month. The component accepts a month as input and outputs a month, status, an order_id, and a customer_id. Both order_id and customer_id are programmatic outputs in the sense that when a user clicks on one of the listed order_id, the order information reflected in other portlets takes the order_id as input.
- `Order Detail Portlet` displays the details of a specific order that includes quantity, status, sku and tracking_id. The component accepts an order_id as input and only tracking_id is output as a programmatic input to other components.
- `Tracking Detail Portlet` displays the tracking information related to a specific order. The component accepts a tracking_id as input. Customer_name is the only programmatic output here.
- `Customer Detail Portlet` displays customer information. The component accepts a customer_id and a customer_name as input and does not provide any programmatic outputs.
- `Account Detail Portlet` displays the account details of a particular order. The component accepts an order_id as input and does not provide any programmatic outputs.



| Match score | WSDL | Location |
|---|---|---|
| 43.7500 | HotSpotFinder.wsdl | http://localhost:8080/thesis/HotSpotFinder.wsdl |
| 40.0000 | AccountDetail.NoOWL.... | http://localhost:8080/thesis/AccountDetail.NoOWL.wsdl |
| 40.0000 | AccountDetail.wsdl | http://localhost:8080/thesis/AccountDetail.wsdl |
| 40.0000 | GoogleMapper.wsdl | http://localhost:8080/thesis/GoogleMapper.wsdl |
| 28.5714 | OrderDetail.NoOWL.wsdl | http://localhost:8080/thesis/OrderDetail.NoOWL.wsdl |
| 28.5714 | OrderDetail.wsdl | http://localhost:8080/thesis/OrderDetail.wsdl |
| 25.0000 | Orders.NoOWL.wsdl | http://localhost:8080/thesis/Orders.NoOWL.wsdl |
| 25.0000 | Orders.wsdl | http://localhost:8080/thesis/Orders.wsdl |
| 12.5000 | CustomerDetail.NoOW... | http://localhost:8080/thesis/CustomerDetail.NoOWL.wsdl |
| 12.5000 | CustomerDetail.wsdl | http://localhost:8080/thesis/CustomerDetail.wsdl |
| 12.5000 | TrackingDetail.NoOWL.... | http://localhost:8080/thesis/TrackingDetail.NoOWL.wsdl |
| 12.5000 | TrackingDetail.wsdl | http://localhost:8080/thesis/TrackingDetail.wsdl |
| 7.1429 | SourceInterface.wsdl | http://localhost:8080/thesis/SourceInterface.wsdl |
| 7.1429 | SourceInterfaceV1.wsdl | http://localhost:8080/thesis/SourceInterfaceV1.wsdl |
| 5.0000 | TargetInterface.wsdl | http://localhost:8080/thesis/TargetInterface.wsdl |
| 5.0000 | TargetInterfaceV1.wsdl | http://localhost:8080/thesis/TargetInterfaceV1.wsdl |

Texas — Select a state
Austin — Select a city

| | | |
|---|---|---|
| Driskill Hotel - Austin TX<br>604 Brazos Street<br>Austin 78701<br>Proximity: 0.2 miles (0.4 km)<br>Map \| Directions | Hotel / Resort | 802.11b Wi-Fi, 802.11g Wi-Fi |
| Fedex Kinko's Downtown<br>327 Congress Ave<br>Austin 78701<br>Proximity: 0.2 miles (0.4 km)<br>Map \| Directions | Store / Shopping Mall | 802.11b Wi-Fi |
| Wyndham Riverside Suites<br>218 College Street<br>San Antonio 78205<br>Proximity: 0.3 miles (0.4 km)<br>Map \| Directions | Hotel / Resort | 802.11g Wi-Fi, 802.11b Wi-Fi, Ethernet |

To create a wire, select a property in the 'Source' component, and drag it onto the desired action in the target component. Context menus are provided for all elements in the diagram.

Include another page | View All Wires... | Hot Spot Finder | Source | City View: cityState | HotSpot Finder: SetLocationCityState | Disable strict type checking | OK | Cancel | Apply

Fig. 2. Sequence of steps in composing an application

### 2.4 Scenario of Assembling Composite Application

Figure 2 illustrates the sequence of screen shots in Lotus Expeditor Client workbench that results in a simple `HotSpotFinder` composite application. Screen A shows the initial composition workbench. The right panel shows the list of components (e.g., `HotSpotFinder`, `GoogleMapper`, `CityStatePicker`, `OrderTracker`) that are available for composition as well as links to other available remote components. The left panel displays all the existing composite applications of a particular user. The middle panel displays the in-progress composite application. To start the composition process, the user must first pick a component from the right panel. The selected component then becomes the search query. When the user clicks on Analyze Composite App menu, a dialog box in screen B is displayed. If there are more than one components on the canvas, the user has the option of choosing "individual" or "merged" matching criteria. If "individual" matching is checked, each of the components' WSDL in the current composite application will be matched individually to the target WSDLs in the repository. If not checked, a merged WSDL file created from all the existing WSDL files in the current composite application will be used in the matching process. After the user enters the desired search criteria at the top of screen and presses the "Find Matches" button, screen C is displayed. By picking the best matched component (the one with the highest score) from the palette and dropping it on the middle panel, screen D is displayed. The middle panel now has two components (`CityStatePicker` and `HotSpotFinder`) which were not aware of each other. At this point, the user can right click on the in-progress composite application which will allow selection of the "wiring" action from the menu. Screen F shows the result of wiring the two selected components on the middle panel. The `CityStatePicker` component (labeled "City View") provides a single output, labeled `cityState`. The `HotSpotFinder` component provides a single input named `SetLocationCityState`. The dotted line indicates that the `cityState` output has been linked to the `SetLocationCityState` input. Therefore, when the output `cityState` is fired, the argument of that output will be sent as the argument to the `SetLocationCityState` input. The composition is now completed and screen G displays the result of running the composed application within the Expeditor Workbench.

## 3 Composite Application Mashup

In this section, we describe the process of creating reusable components and the techniques, technologies, and algorithms that can be leveraged to provide progressive mashup of composite applications.

### 3.1 Building Components

In order to create a component for reuse in a composite application, a few additional steps are required beyond what is necessary to create a stand-alone application or component. However, the majority of the process is similar to creating standard applications. Obviously, different component frameworks will advocate slightly different protocols for component creation. In J2EE-based component systems, the concrete processes and tools used for developing and deploying a component will be different from Lotus Expeditor. However the fundamental principles are the same. One of the first steps that any developer will need to do when building a component for an application is to decide on the graphical interface technology. This may be based on Java Swing, servlet, AWT, native widgets, portlets, and HTML. The second step is to locate the data that is to be presented in the graphical user interface. Often this information consists of records from a relational database or some other back-end data sources. The third step is to write a bit of code to deal with the programmatic input and the output of the component. Up to this point there is no difference in the process between creating components for a composite application and creating a component for a standard application. For components to be reusable in a composition framework, a few additional steps must be taken. The first additional step is to decide which programmatic inputs, outputs, and operations should be exposed if this component is reusable. The second additional step is to generate a WSDL file that describes the exposed programmatic input, output, and operations of the component. The last additional step is to annotate the generated WSDL file with semantic information. We assume that there are existing ontological models that we can use. Otherwise, those ontological models must be created first. The following are the steps involved for building the `CityStatePicker` component using the Lotus Expeditor Toolkit:

- Develop `CityStatePickerView`, a Java class which extends Eclipse UI Extension class. This class is responsible for creating the UI to be displayed to a user in Lotus Expeditor Rich Client. The UI should allow her to select a state and then a city within that state. When the city is selected, the city and state information should be broadcasted to the `PropertyBroker` in Expeditor.
- Develop a Java class, `pubCityState` to broadcast UI events to the `PropertyBroker` in Expeditor. This is a helper class for `CityStatePickerView`.
- Create a WSDL, `CityStatePicker.wsdl`, to expose the properties (UI inputs) and actions (operations) associated with the `CityStatePicker` that can be reused. This file is created using properties wizard and the editor available in the Expeditor Toolkit in our framework.
- Annotate the generated WSDL file with semantic information on any elements in the WSDL file that will play a part in matching the component.
- Create a manifest file for automatically deploying the component to OSGi compatible Lotus Expeditor middleware.

With the wide availability of Web service development toolkits, the generation of WSDL file can be automatic. The semantic annotation of WSDL, however, has to be done manually. Currently, we use a plain text editor for performing the annotation. However, we can envision development of a simple graphical tool to simplify the annotation process such as the semantic annotation tool available in Kepler system [10].

## 3.2 Semantic Annotation of Components

The WSDL-based programmatic inputs and outputs of a reusable component are annotated using suitable ontological models to enable searching using higher level concepts. We choose to use SAWSDL annotation scheme because SAWSDL is currently a recommended W3C standard. This standard does not dictate a language for representing the specific ontology. It defines a method of including ontological information written in any specification languages into a WSDL document. In this mashup tool, we chose to use ontological model specified in OWL for annotation. This allows us to leverage the established semantic web matching techniques developed in OWL for flexible and intelligent search for compatible components. In this sense, WSDL+semantic annotation (SAWSDL) is used as a loose standardization of APIs that can be exposed by the diverse kinds of components that we want to mashup.

However, using SAWSDL prevents us from including the emerging *annotated* RESTful Web services that adhere to the simpler REST architecture style of service invocation in our mashup tool. RESTful Web services do not have associated WSDL files. Instead, they are annotated using a different scheme called SA-REST [11] which is based on HTML and RDFa[1]. Despite its simplicity, SA-REST in its current form is not suitable for use in the Lotus Expeditor framework. The annotated CAE (Composite Application Editor) is dependent on WSDL format for data mediation. Moreover, adopting SA-REST will require significant changes to the structural matching part of the semantic Web matching algorithm. However, as described in [12], it is feasible to translate SA-REST into SAWSDL. Thus RESTful Web services with annotation can be accommodated in our framework with some additional effort. Within Lotus Expeditor framework, annotating components using either SAWSDL or SA-REST will have similar limitations when it comes to composing applications. This is because both require the components to be annotated manually and a-priori. If a particular capability of a component is not being annotated, at runtime, it is impossible to leverage that capability for mashup even if it is useful to utilize that capability within a composite application.

Semantic annotations (SAWSDL) only work if there is a unified ontological model. If the ontological model for the component we are interested in annotating is not available, it must be created first. Tools such as the Protege-OWL [13] editor can be used to create OWL-based ontological models. However, a given enterprise may have a collection of models that already exist to describe the data and processes used in the enterprise. Further, a given industry may have a collection of models that have been already created to describe the unique characteristics of that industry. If the component being built is intended for use in a given enterprise or industry, care should be taken to use existing ontological models where it makes sense. Since the SAWSDL specification allows multiple models to be attached to a given element, it may be appropriate to provide one or more enterprise, industry, and custom models to a particular element in the component. The different ontological models must all be represented in OWL formalism

1. http://www.w3.org/TR/xhtml-rdfa-primer/

in our framework to leverage using existing semantic Web matching techniques.

Figure 3 is a WSDL file for `CityStatePicker` component. Lines 5 and 9 import the necessary namespaces used to add the semantic annotations. Lines 15-17 define a new message named “`cityState`”, which defines the name of the string that will be output when a user interacts with this component. Line 16 describes a message that has been annotated with a reference to an element in an ontological model. This is shown as `wssem:modelReference="TravelOnt#City"` in the WSDL file. `TravelOnt` is an OWL-based ontological model that is available on the Web. With this annotation, we are describing the message in terms of an OWL class in an existing ontological model. This message is set as an output of the “`pubCityState`” operation in a portType (lines 18-22) and included in a portlet type binding (lines 23-35). The “`pubCityState`” operation corresponds to broadcasting the city and state information to `PropertyBroker`.

With this markup using the standard grammar defined by WSDL and SAWSDL, the component’s output can now be matched against other components’ programmatic inputs using existing semantic Web service matching technologies. Note that only properties that are pertinent to reuse are present in the WSDL file. By including the annotation, the matching engine is able to match based on capabilities of the component as described in the ontological model. For example, assume the “`cityState`” part element was to be compared against another component’s part element “`county`”. The text-based matching would not count these as a possible match because the two strings are not equal (i.e. “`cityState`” != “`county`”). However, if the “`county`” element had a semantic annotation of “`TravelOnt#County`” in its *modelReference* attribute, the matching logic would be able to compare the model types City and County. If the model described a relationship between a City and a County, perhaps using the *hasProperty* OWL attribute, it could be determined that a city is in a county and both are part of a state. Thus, the two elements will return a matching score because the analysis would show that these two elements are very closely related.

```
01  <?xml version=“1.0” encoding=“UTF-8”?>
02  <definitions name=“edu.txstate.mpc”
03     targetNamespace=“http://www.ibm.com/wps/c2a/edu.txstate.mpc”
04     xmlns=“http://schemas.xmlsoap.org/wsdl/”
05     xmlns:TravelOnt=“http://localhost:8080/thesis/travel.owl”
06     xmlns:portlet=“http://www.ibm.com/wps/c2a”
07     xmlns:soap=“http://schemas.xmlsoap.org/wsdl/soap/”
08     xmlns:tns=“http://www.ibm.com/wps/c2a/edu.txstate.mpc”
09     xmlns:wssem=“http://www.w3.org/ns/sawsdl”
10     xmlns:xsd=“http://www.w3.org/2001/XMLSchema”
11     xmlns:xsi=“http://www.w3.org/2001/XMLSchema-instance”>
12     <types>
13       <xsd:schema targetNamespace=“http://www.ibm.com/wps/c2a/edu.txstate.mpc”/>
14     </types>
15     <message name=“cityState”>
16     <part name=“cityState” type=“xsd:string” wssem:modelReference=“TravelOnt#City”/>
17     </message>
18     <portType name=“edu.txstate.mpc_Service”>
19       <operation name=“pubCityState”>
20         <output message=“tns:cityState”/>
21       </operation>
22     </portType>
23     <binding name=“edu.txstate.mpcbinding” type=“tns.edu.txstate.mpc_Service”>
24       <portlet:binding/>
25       <operation name=“pubCityState”>
26         <portlet:action activeOnStartup=“true” caption=“pubCityState”
27           description=“Announces the city and state” name=“pubCityState”
28           selectOnMultipleMatch=“false” type=“standard”/>
29         <output>
30           <portlet:param boundTo=“request-attribute” caption=“cityState”
31             description=“Published the city/state selected”
32             name=“cityState” partname=“cityState”/>
33         </output>
34       </operation>
35     </binding>
36  </definitions>
```

Fig. 3. WSDL with semantic markup for `CityStatePicker` component

## 3.3 Semantic Matching of Components

Describing the semantic knowledge of a component via the annotation process described above has a unique advantage in that the additional metadata added to the component’s WSDL do not change the structure of the component’s description. Therefore, existing semantic Web services matching technologies and algorithms can be used directly for matching the application components. One such set of algorithms, adopted in our work, is described by T. Syeda-Mahmood [14], which combines the use of semantic (domain independent) and ontological (domain dependent) matching for the purpose of matching Web service components. In the following, we will briefly illustrate the matching mechanism. Interested readers are referred to [14] for more details.

**Domain Independent Matching.** The input to this matching algorithm is a single WSDL file and the collection of target WSDL files to match upon. The output is a target WSDL file that has the largest ratio of matched attributes (matching score) or a list of target WSDL files sorted by the ratio of matched attributes. In order to calculate the matching score, both the query and the target WSDL files must be pre-processed using the following procedures:

- *Word tokenization*: By exploiting cues such as changes in font and presence of delimiter, a multi-term word attribute is tokenized.
- *Part-of-speech tagging and filtering*: Simple grammar rules are employed to label tokenized attributes. Stop words are removed.
- *Abbreviation expansion*: Both domain-independent and domain-specific vocabularies are used to expand tokenized words that is abbreviated. For example, *zip* is expanded into *zipcode*.
- *Association of tag type with attributes*: Each tokenized attribute is labelled with its specific WSDL element types. For example, *name* in Figure 3 is labeled with “*part*” tag.
- *Synonym search*: A thesaurus, in this case WordNet [15], is used to construct a list of synonyms for each tokenized attribute. For example, the word “*town*” is included as a synonym for a tokenized attribute “*city*”.

Given a pair of matching attributes ($\mathcal{A}$, $\mathcal{B}$) with $\mathcal{A}$ equal to [$name$=`cityState`] and $\mathcal{B}$ equal to [$name$=`county`], the similarity matching score of this pair of attributes is calculated based on the following formula:

$$Sem(\mathcal{A}, \mathcal{B}) = 2 * Match(\mathcal{A}, \mathcal{B})/(m + n) \tag{1}$$

Where $m$ and $n$ are the number of valid tokens in $\mathcal{A}$ and $\mathcal{B}$.

Both $\mathcal{A}$ and $\mathcal{B}$ must be of the same structural type. In this case they both must be tagged with *Part* element of a WSDL file. $Match(\mathcal{A}, \mathcal{B})$ returns the number of matching tokens. Given a query WSDL file and a collection of target WSDL files to match upon, the best matched WSDL file is the one that has the highest $Sem(\mathcal{A}, \mathcal{B})$ score.

| Attribute Pair | Relationship | Distance Score |
|---|---|---|
| ($\mathcal{A}$, $\mathcal{B}$) | EquivalentClass | 0.0 |
| ($\mathcal{A}$, $\mathcal{B}$) | HasPropertyClass | 0.3 |
| ($\mathcal{A}$, $\mathcal{B}$) | HasPartClass | 0.5 |
| ($\mathcal{A}$, $\mathcal{B}$) | SubClassOf | 0.7 |
| ($\mathcal{A}$, $\mathcal{B}$) | Other | 1.0 |

TABLE 1
A simple distance scoring scheme

**Domain Dependent Matching.** Domain independent matching described above is basically an enhanced keyword-based matching. Domain dependent matching makes use of ontological relationship between tokenized attributes. Only attributes that are being annotated in the WSDL files can be used for domain dependent match. The semantic for each attribute are compared using a custom ontology matching algorithm from SNOBASE (Semantic Network Ontology Base), an IBM ontology management framework for loading ontologies from files and via the Internet [16]. This algorithm takes into account the relationships between the given attributes, such as *inheritance*, *hasPart*, *hasProperty*, and *equivalent* classes. A simple distance scoring scheme as shown in Table 1 is used. This scoring scheme gives a coarse indication of semantic distance between concepts. For example, the distance score for two attributes that have an *equivalent* relationship is 0. For the inheritance relationship (i.e., *SubClassOf* in Table 1), the score is 0.7. The matching score between a pair of matching services $\mathcal{S}_q$ and $\mathcal{S}_i$ is calculated using the following formula:

$$Match(\mathcal{S}_q, \mathcal{S}_i) = 2 * h_i * \sum_j (1 - dist(i,j))/(n_i + n_q) \quad (2)$$

Where $n_i$ is the number of attributes of the query service $\mathcal{S}_q$ and $n_i$ is the number of annotated attributes present in service $\mathcal{S}_i$, $h_i$ is the number of annotated attributes of services $\mathcal{S}_i$ that have been matched out of $n_q$, and finally $dist(i,j)$ is ontological distance score between the $j^{th}$ term in service $\mathcal{S}_i$ and a corresponding query term. The best matched WSDL file to a query WSDL file is given by the one that has the highest ontological match score.

**Final Score.** It is calculated using a winner-takes-all approach. The maximum of the domain independent score and domain dependent score is reported as the overall matching score. Currently, the higher score is taken to mean a "better match". The advantage of using two different scoring mechanisms in one framework is that the same matching algorithm is applicable to either annotated or un-annotated WSDL files. If WSDLs are all marked up consistently, the algorithm will be able to find more accurate match based on semantic information. If none of the WSDLs is annotated, the domain independent matching can still be applied to return the "matches".

**Discussions.** We reused much of the same matching logic and algorithms from semantic Web services. However, there are several fundamental differences when dealing with non-Web services based components. In many cases, when composing with Web service components, a developer is looking for APIs that can either:

- *Match*, i.e., using the output from a single Web service and finding a second Web service that can take it as input. The developer can continue this process and string together several Web services choreographed by a specific process model. This is typically referred to as *process-based Web service composition* [17], [18], [19], or
- *Compose*, i.e., starting with a known output and a known input, through some intelligent search/inference techniques, the system returns one or more services that will transform the output of the first Web service into something that can be consumed by the final Web service. This is typically referred to as *dynamic semantic Web service composition* [20], [21], [22].

The difference with respect to our composite applications is that in most cases the goal is not to put together a single business process or tightly link fragments of software processes for the purpose of automating a specific task; rather, the goal is to integrate separately created components together "on the glass" [23] and provide the ability for those applications to communicate or interact without prior knowledge of each other or in any specific order. There is no explicit control-flow specified between the communicating components. A component can start execution whenever it receives the required input. This data-flow oriented paradigm of composition made this framework suitable for users who do not have knowledge of control-flow constructs in programming or process languages to compose their applications on the fly. Furthermore, users do not need to learn how to invoke application specific APIs in order to compose different applications.

There is no specific begin and end state in our composition framework. There is also no specific ordering in terms of the composition process. For example, the next component to be composed does not have to depend on the previous component. It depends on what components are already in the application that we are composing. We advocate progressive composition process that allows a user to explore or preview the functionalities of the composite application iteratively and refine them at any point in time. Users have choices on what to compose that will adapt to their working style. For example, when composing an `OrderTracking` composite application, if the user is a salesperson, the application will include the `OrderPortlet` (Figure 11). However, if the user is a customer, it is not appropriate to include the `OrderPortlet`.

## 4 Experimental Study

The experiments reported in this section were completed using the sample applications described in Section 2.3. The experiments were conducted on a Lenovo ThinkPad T60p running Microsoft WindowsXP SP2. An Apache HTTP server in conjunction with an IBM WebSphere Portal Server 6.0 was used to simulate the component library. In order to make it easy to drive the test cases and to analyze the results, a graphical user interface component was created (see Figure 4).

This component reads the currently executing composite application and drives the test cases. The “View Filter” and “Display Filter” sections allow the user to set the graphical user interface search criteria. The main panel of the window displays the score associated with each of the target WSDLs that was included in a search request. The “Find Matches” button starts the search process. Finally, the “Use individual matching” checkbox allows the user to specify which type of matching will be used. If checked, each of the components’ WSDLs in the current composite application will be matched individually to the target WSDLs in the repository. Otherwise, a “merged” matching will be used.

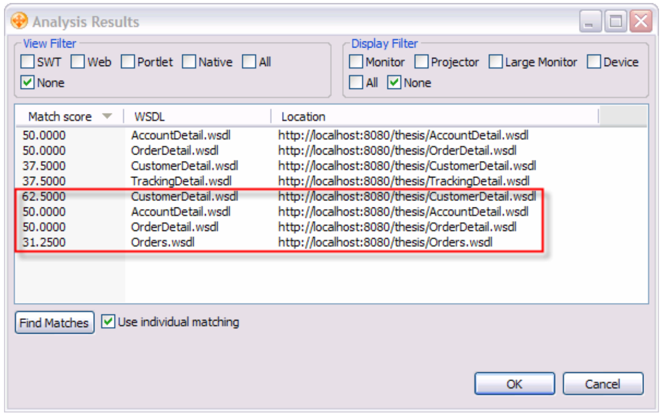
In addition to the application components required for the composition of the two applications described in Section 2.3, an additional four WSDLs with semantic annotations were included in the target component repository. These WSDLs are SourceInterface.wsdl, SourceInterfaceV1.wsdl, TargetInterface.wsdl, and TargetInterfaceV1.wsdl. These additional WSDLs were added to the repository to simulate other components that should not be matched in the context of our composite applications. These WSDLs describe components for a retail order system, and are not applicable as components in the two applications being mashed up using our framework.

## 4.1 Experiment One - Basic Matching

The first experiment shows simple matching using two component WSDLs and the semantic web matching logic. The input for this scenario is the CityStatePicker.wsdl and the target is the HotSpotFinder.wsdl. When run through the matching logic, a score of 50 is produced. This is a reasonable score because of the differences in the two WSDLs. The CityStatePicker.wsdl file defines a single message, *cityState*, and the HotSpotFinder.wsdl defines two messages, *city* and *address*. In order to show the impact of the semantic matching only, a modified version of the HotSpotFinder.wsdl is used. In the modified version, the identifying names such as *city* and *address* are replaced with random strings such as *vvv* and *ddd*. Because these do not match fields in the CityStatePicker.wsdl, the ontological score is always returned. The resultant score in this case is 37.50. This lower score can be accounted for based on the fact that only the message elements have been annotated with additional semantics. Lastly, we remove all the semantic annotation in the WSDL documents so that only pure keyword matching can be used. In this run, the matching score drops to 25. Additional changes to the WSDL that remove identifying city and state keywords while preserving its functionality cause the score to drop even more. This shows that the semantic matching algorithm is working as expected and that we have a valid environment for conducting other experiments.

## 4.2 Experiment Two - Merged WSDL Matching

The second experiment shows the effect of using a merged WSDL to find compatible components for a composite application. The two inputs for this experiment are Orders.wsdl and TrackingDetail.wsdl. These are matched against the other three target WSDLs that are part of the second Order Tracking



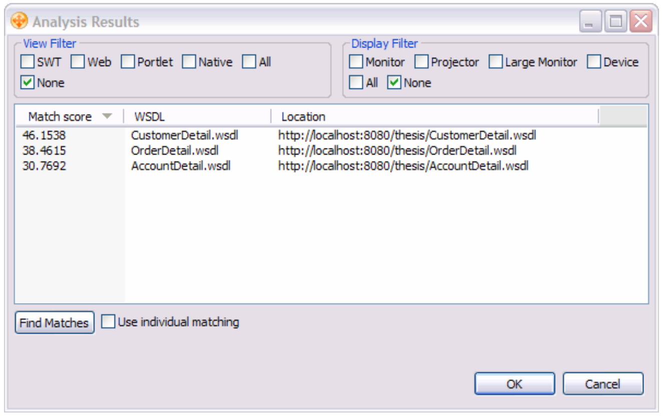
View Filter: SWT [ ] Web [ ] Portlet [ ] Native [ ] All [ ] [✓ None]

Display Filter: Monitor [ ] Projector [ ] Large Monitor [ ] Device [ ] All [ ] [✓ None]

| Match score | WSDL | Location |
|---|---|---|
| 50.0000 | AccountDetail.wsdl | http://localhost:8080/thesis/AccountDetail.wsdl |
| 50.0000 | OrderDetail.wsdl | http://localhost:8080/thesis/OrderDetail.wsdl |
| 37.5000 | CustomerDetail.wsdl | http://localhost:8080/thesis/CustomerDetail.wsdl |
| 37.5000 | TrackingDetail.wsdl | http://localhost:8080/thesis/TrackingDetail.wsdl |
| 62.5000 | CustomerDetail.wsdl | http://localhost:8080/thesis/CustomerDetail.wsdl |
| 50.0000 | AccountDetail.wsdl | http://localhost:8080/thesis/AccountDetail.wsdl |
| 50.0000 | OrderDetail.wsdl | http://localhost:8080/thesis/OrderDetail.wsdl |
| 31.2500 | Orders.wsdl | http://localhost:8080/thesis/Orders.wsdl |

[✓ Use individual matching]

Fig. 4. Results of individual matching in experiment 2



View Filter: SWT [ ] Web [ ] Portlet [ ] Native [ ] All [ ] [✓ None]

Display Filter: Monitor [ ] Projector [ ] Large Monitor [ ] Device [ ] All [ ] [✓ None]

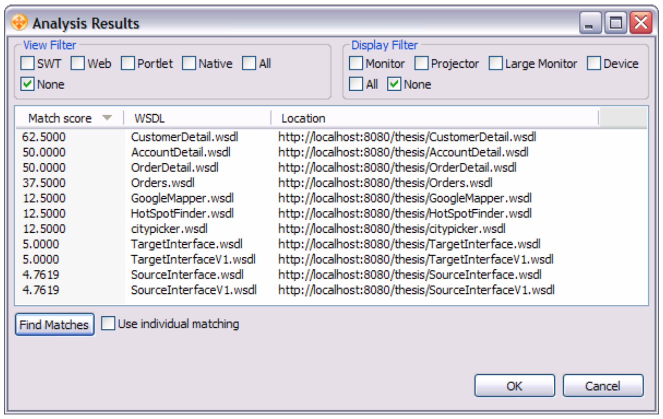
| Match score | WSDL | Location |
|---|---|---|
| 46.1538 | CustomerDetail.wsdl | http://localhost:8080/thesis/CustomerDetail.wsdl |
| 38.4615 | OrderDetail.wsdl | http://localhost:8080/thesis/OrderDetail.wsdl |
| 30.7692 | AccountDetail.wsdl | http://localhost:8080/thesis/AccountDetail.wsdl |

[ ] Use individual matching

Fig. 5. Results of merged matching in experiment 2

application described in Section 2.3, specifically AccountDetail.wsdl, OrdersDetail.wsdl, and CustomerDetails.wsdl. Given this setup, any of the three target WSDLs could be a good match because each of them have inputs that can be satisfied by the available outputs of the `Orders` and `TrackingDetails` components. In this experiment, we first match the two input components individually against the other three. The results are shown in Figure 4. The first four entries in the list are the result of matching against the Orders.wsdl; the second four results, grouped in the box, are the result of matching against the TrackingDetails.wsdl. As we can see, the `CustomerDetail` component has the highest overall match value of the possible choices. This makes sense because the single output of the `TrackingDetails` component matches one of the two inputs to the `CustomerDetails` component. When we look at the results for the `Orders` component, we see the `CustomerDetail` component is ranked lower than either the `AccountDetail` or the `OrderDetail` component and equally scored against the `TrackingDetails` component. The `AccountDetail` and `OrderDetail` components also scored fairly well in the match against the `TrackingDetails` component.

Given this ambiguity, how do we decide which component to add to the composite application? This is where the merged WSDL search can assist. If we do a merge WSDL search, we combine the inputs and outputs of the two given components and match those against the remaining components in the



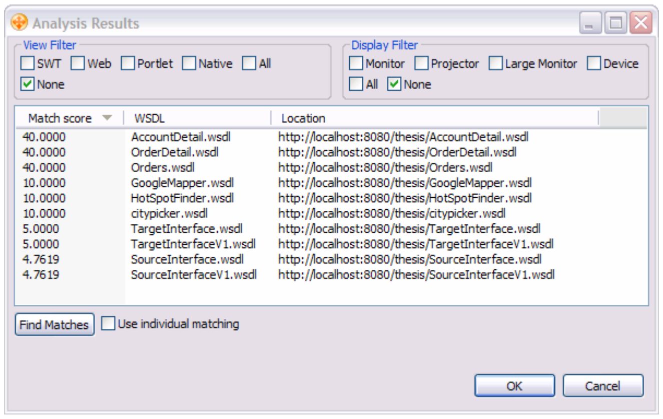
View Filter: ☐ SWT ☐ Web ☐ Portlet ☐ Native ☐ All ✓ None

Display Filter: ☐ Monitor ☐ Projector ☐ Large Monitor ☐ Device ☐ All ✓ None

| Match score | WSDL | Location |
|---|---|---|
| 62.5000 | CustomerDetail.wsdl | http://localhost:8080/thesis/CustomerDetail.wsdl |
| 50.0000 | AccountDetail.wsdl | http://localhost:8080/thesis/AccountDetail.wsdl |
| 50.0000 | OrderDetail.wsdl | http://localhost:8080/thesis/OrderDetail.wsdl |
| 37.5000 | Orders.wsdl | http://localhost:8080/thesis/Orders.wsdl |
| 12.5000 | GoogleMapper.wsdl | http://localhost:8080/thesis/GoogleMapper.wsdl |
| 12.5000 | HotSpotFinder.wsdl | http://localhost:8080/thesis/HotSpotFinder.wsdl |
| 12.5000 | citypicker.wsdl | http://localhost:8080/thesis/citypicker.wsdl |
| 5.0000 | TargetInterface.wsdl | http://localhost:8080/thesis/TargetInterface.wsdl |
| 5.0000 | TargetInterfaceV1.wsdl | http://localhost:8080/thesis/TargetInterfaceV1.wsdl |
| 4.7619 | SourceInterface.wsdl | http://localhost:8080/thesis/SourceInterface.wsdl |
| 4.7619 | SourceInterfaceV1.wsdl | http://localhost:8080/thesis/SourceInterfaceV1.wsdl |

☐ Use individual matching

Fig. 6. Matching with TrackingDetail



View Filter: ☐ SWT ☐ Web ☐ Portlet ☐ Native ☐ All ✓ None

Display Filter: ☐ Monitor ☐ Projector ☐ Large Monitor ☐ Device ☐ All ✓ None

| Match score | WSDL | Location |
|---|---|---|
| 40.0000 | AccountDetail.wsdl | http://localhost:8080/thesis/AccountDetail.wsdl |
| 40.0000 | OrderDetail.wsdl | http://localhost:8080/thesis/OrderDetail.wsdl |
| 40.0000 | Orders.wsdl | http://localhost:8080/thesis/Orders.wsdl |
| 10.0000 | GoogleMapper.wsdl | http://localhost:8080/thesis/GoogleMapper.wsdl |
| 10.0000 | HotSpotFinder.wsdl | http://localhost:8080/thesis/HotSpotFinder.wsdl |
| 10.0000 | citypicker.wsdl | http://localhost:8080/thesis/citypicker.wsdl |
| 5.0000 | TargetInterface.wsdl | http://localhost:8080/thesis/TargetInterface.wsdl |
| 5.0000 | TargetInterfaceV1.wsdl | http://localhost:8080/thesis/TargetInterfaceV1.wsdl |
| 4.7619 | SourceInterface.wsdl | http://localhost:8080/thesis/SourceInterface.wsdl |
| 4.7619 | SourceInterfaceV1.wsdl | http://localhost:8080/thesis/SourceInterfaceV1.wsdl |

☐ Use individual matching

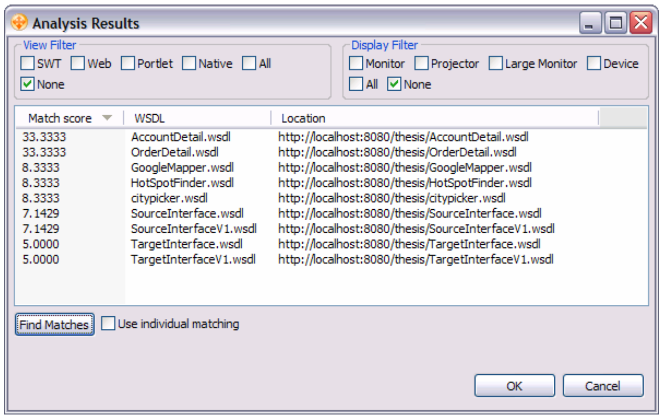
Fig. 7. Matching with TrackingDetail and CustomerDetail

catalog. The results of this search are shown in Figure 5. In this case, we can now see that the `CustomerDetail` component is probably the best component to add to the composite application.

### 4.3 Experiment Three - Assembling the Order Tracking Composite Application

We have built a number of composite applications using our mashup platform. In this experiment, we will particularly show how the order tracking composite application can be built. The goal is to demonstrate that our mashup framework can be used for components that were built using different component technologies. Here, all the components were originally built as portlets. From looking at the possible starting points, the `Orders` component would be the most obvious one to use since it contains several outputs. However, instead we will use the `TrackingDetail` component to show how we can build the complete application. This is a reasonable choice to begin with because we are building an order tracking application. The first step is to add the `TrackingDetail` component to the application and run the search. The results, as shown in Figure 6, tell us that the `CustomerDetail` component would be a good one to add at this point.

Once the `CustomerDetail` component is added to the application, we can run the analysis again. Figure 7 shows the results of this process. Looking at the scores, the average scores have decreased, though only by ten points. You will



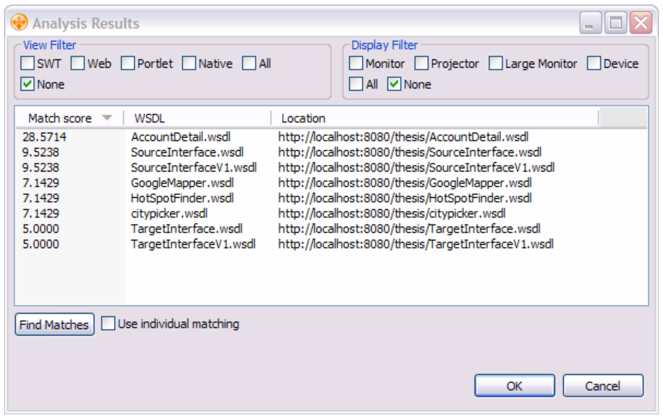
View Filter: ☐ SWT ☐ Web ☐ Portlet ☐ Native ☐ All ✓ None

Display Filter: ☐ Monitor ☐ Projector ☐ Large Monitor ☐ Device ☐ All ✓ None

| Match score | WSDL | Location |
|---|---|---|
| 33.3333 | AccountDetail.wsdl | http://localhost:8080/thesis/AccountDetail.wsdl |
| 33.3333 | OrderDetail.wsdl | http://localhost:8080/thesis/OrderDetail.wsdl |
| 8.3333 | GoogleMapper.wsdl | http://localhost:8080/thesis/GoogleMapper.wsdl |
| 8.3333 | HotSpotFinder.wsdl | http://localhost:8080/thesis/HotSpotFinder.wsdl |
| 8.3333 | citypicker.wsdl | http://localhost:8080/thesis/citypicker.wsdl |
| 7.1429 | SourceInterface.wsdl | http://localhost:8080/thesis/SourceInterface.wsdl |
| 7.1429 | SourceInterfaceV1.wsdl | http://localhost:8080/thesis/SourceInterfaceV1.wsdl |
| 5.0000 | TargetInterface.wsdl | http://localhost:8080/thesis/TargetInterface.wsdl |
| 5.0000 | TargetInterfaceV1.wsdl | http://localhost:8080/thesis/TargetInterfaceV1.wsdl |

☐ Use individual matching

Fig. 8. Matching with TrackingDetail, CustomerDetail, and Orders



View Filter: ☐ SWT ☐ Web ☐ Portlet ☐ Native ☐ All ✓ None

Display Filter: ☐ Monitor ☐ Projector ☐ Large Monitor ☐ Device ☐ All ✓ None

| Match score | WSDL | Location |
|---|---|---|
| 28.5714 | AccountDetail.wsdl | http://localhost:8080/thesis/AccountDetail.wsdl |
| 9.5238 | SourceInterface.wsdl | http://localhost:8080/thesis/SourceInterface.wsdl |
| 9.5238 | SourceInterfaceV1.wsdl | http://localhost:8080/thesis/SourceInterfaceV1.wsdl |
| 7.1429 | GoogleMapper.wsdl | http://localhost:8080/thesis/GoogleMapper.wsdl |
| 7.1429 | HotSpotFinder.wsdl | http://localhost:8080/thesis/HotSpotFinder.wsdl |
| 7.1429 | citypicker.wsdl | http://localhost:8080/thesis/citypicker.wsdl |
| 5.0000 | TargetInterface.wsdl | http://localhost:8080/thesis/TargetInterface.wsdl |
| 5.0000 | TargetInterfaceV1.wsdl | http://localhost:8080/thesis/TargetInterfaceV1.wsdl |

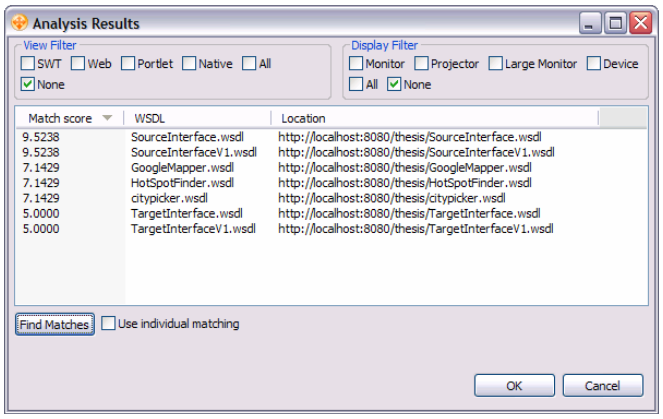
☐ Use individual matching

Fig. 9. Matching with TrackingDetail, CustomerDetail, Orders, and OrderDetail

also observe that the score for the `Orders` component has actually increased by a small amount. Given that there are three possible components to choose from with equal scores, we will choose the `Orders` component because it has shown a consistent increase in value over the last two searches.

The `Orders` component is added to the application and the matching analysis is run again. This time, in Figure 8, we observe that the overall scores have decreased again, but there is still a significant difference between the two highest scores and the third score. There is no real drive to choose one component over the other based on the scores, so we need to choose one. Because we are building an order tracking application, the name `OrderDetail` seems like a better choice than `AccountDetail`. In other experiments not detailed here, it was seen that choosing the `AccountDetail` component eventually lead to the same final composite application described in this section. Additionally, the iterative nature of composite application assembly allows assemblers to try out components and remove them if they do not prove to be useful. In this case, if the `AccountDetail` was found not to be usable in the application, the assembler could remove it and instead add the `OrderDetail` component.

We add the `OrderDetail` to the application and run the analysis again. As we see in Figure 9, the top score has again dropped, but it is significantly higher than the other scores. We therefore decide to add the `AccountDetail` component.



| Match score | WSDL | Location |
|---|---|---|
| 9.5238 | SourceInterface.wsdl | http://localhost:8080/thesis/SourceInterface.wsdl |
| 9.5238 | SourceInterfaceV1.wsdl | http://localhost:8080/thesis/SourceInterfaceV1.wsdl |
| 7.1429 | GoogleMapper.wsdl | http://localhost:8080/thesis/GoogleMapper.wsdl |
| 7.1429 | HotSpotFinder.wsdl | http://localhost:8080/thesis/HotSpotFinder.wsdl |
| 7.1429 | citypicker.wsdl | http://localhost:8080/thesis/citypicker.wsdl |
| 5.0000 | TargetInterface.wsdl | http://localhost:8080/thesis/TargetInterface.wsdl |
| 5.0000 | TargetInterfaceV1.wsdl | http://localhost:8080/thesis/TargetInterfaceV1.wsdl |

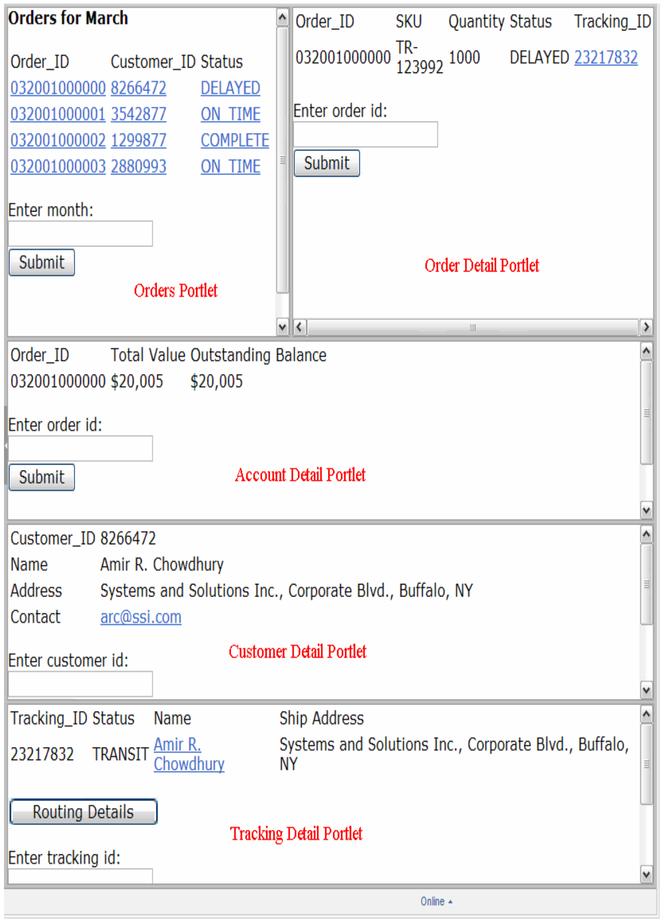
Fig. 10. Matching with All Order Detail components

With the `AccountDetail` component added to the application, we run the analysis again and get the results as shown in Figure 10. The top ranking score is now 9.5238 - much lower than what we started with and also much lower than the last component we added. It is reasonable to assume that the application is now complete. We can now customize the layout of the application to suit the user's needs based on the component we have selected. Once a tracking ID is entered in "Tracking Detail Portlet", the related information will be shown immediately in other components. Figure 11 shows the assembled order tracking composite application.

## 4.4 Experiment Four - Effectiveness and Scalability of the Semantic Matching

We also conducted experiments to study the effectiveness and scalability of our proposed semantic matching process. To test the effectiveness of the semantic matching, we added additional 11 new WSDL files. These WSDLs were downloaded from public domain Web service portals [24], [25]. These WSDL files are much more complex in the sense that they all have multiple messages, operations and bindings as compared to WSDLs generated for the GUI components used in experiments 1-3. Without any annotations, all these new Web services (shown in italics font) have low matching scores as shown in Figure 12. However, when we annotated the message for the dictionary.wsdl with TravelOnt#City (i.e. the same ontology that the source CityStatePicker component is using), the score increases from 0.0 to 14.28. Figure 13 shows the resulting scores of running with annotated dictionary WSDL (we renamed the name of the Dictionary.wsdl file for this experiment). Note that these experiments were run in batch mode for efficiency purpose and thus the results are not displayed in a GUI-based screen as in other experiments. Interested readers are referred to [14] for a more detailed discussion on the effectiveness of semantic matching.

To test the scalability of the matching algorithm, we logged the elapsed time when matching with different number of target WSDLS. We started with randomly selected 11 WSDLs (used in the previous experiment) and doubling the number of WSDLS in the target for each additional run. Thus, we ran this experiment with 11, 22, 44, 88 and 166 WSDLs. The memory heap size is set to 1G. Figure 14 shows that as the number of WSDLs increases exponentially, the time it takes



Orders for March

| Order_ID | Customer_ID | Status |
|---|---|---|
| 032001000000 | 8266472 | DELAYED |
| 032001000001 | 3542877 | ON_TIME |
| 032001000002 | 1299877 | COMPLETE |
| 032001000003 | 2880993 | ON_TIME |

Enter month:

| Order_ID | SKU | Quantity | Status | Tracking_ID |
|---|---|---|---|---|
| 032001000000 | TR-123992 | 1000 | DELAYED | 23217832 |

Enter order id:

| Order_ID | Total Value | Outstanding Balance |
|---|---|---|
| 032001000000 | $20,005 | $20,005 |

Enter order id:

| | |
|---|---|
| Customer_ID | 8266472 |
| Name | Amir R. Chowdhury |
| Address | Systems and Solutions Inc., Corporate Blvd., Buffalo, NY |
| Contact | arc@ssi.com |

Enter customer id:

| Tracking_ID | Status | Name | Ship Address |
|---|---|---|---|
| 23217832 | TRANSIT | Amir R. Chowdhury | Systems and Solutions Inc., Corporate Blvd., Buffalo, NY |

Enter tracking id:

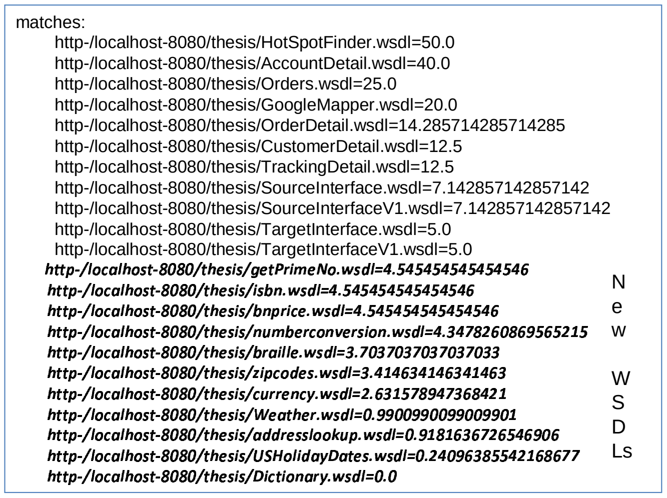
Fig. 11. Assembled OrderTracking Composite Application

for completing the match increases in a linear fashion. This demonstrates that the matching algorithm is scalable. The only limiting factor is the availability of heap memory size of the computer where the system is running.

## 4.5 Experiment Analysis

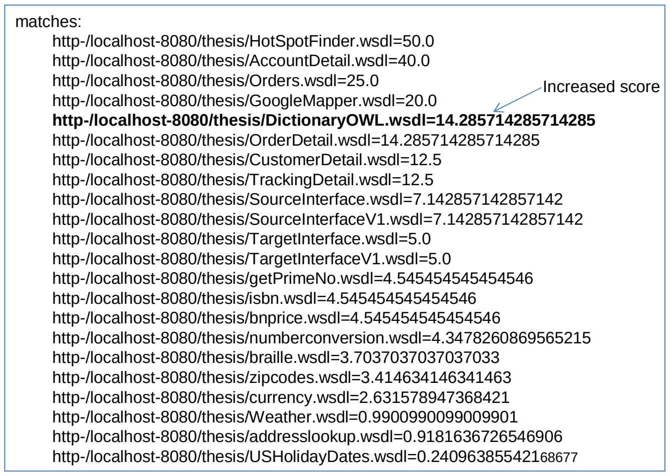
Experiment one and two have shown that the existing Web services matching code can be used in conjunction with composite applications. Because the matching logic uses both text-based matching and semantic matching, the matching algorithm can be used without adding the semantic markup. However, as we observed in experiment one, the semantic matching always provides better results. For example, when two components are named differently yet provide the same functionality, the semantic matching is able to find the match.

Experiments three has shown that it is possible to build a composite application from a collection of different components—implemented using different technologies—using semantic annotations and semantic Web service matching logic. While there is no automatic way to start the building process, once a starting point is selected, the remaining compatible components begin to stand out in the repository searches using semantic-based mashup. As with the assembly of the `OrderTracking` application, the components that could be added to this application really stood out with scores three or more times greater for components that were not appropriate. The artificial `Source` and `Target` WSDLs that were added to the repository continue to score low in all cases.



```
matches:
     http-/localhost-8080/thesis/HotSpotFinder.wsdl=50.0
     http-/localhost-8080/thesis/AccountDetail.wsdl=40.0
     http-/localhost-8080/thesis/Orders.wsdl=25.0
     http-/localhost-8080/thesis/GoogleMapper.wsdl=20.0
     http-/localhost-8080/thesis/OrderDetail.wsdl=14.285714285714285
     http-/localhost-8080/thesis/CustomerDetail.wsdl=12.5
     http-/localhost-8080/thesis/TrackingDetail.wsdl=12.5
     http-/localhost-8080/thesis/SourceInterface.wsdl=7.142857142857142
     http-/localhost-8080/thesis/SourceInterfaceV1.wsdl=7.142857142857142
     http-/localhost-8080/thesis/TargetInterface.wsdl=5.0
     http-/localhost-8080/thesis/TargetInterfaceV1.wsdl=5.0
    http-/localhost-8080/thesis/getPrimeNo.wsdl=4.545454545454546
    http-/localhost-8080/thesis/isbn.wsdl=4.545454545454546
    http-/localhost-8080/thesis/bnprice.wsdl=4.545454545454546
    http-/localhost-8080/thesis/numberconversion.wsdl=4.3478260869565215
    http-/localhost-8080/thesis/braille.wsdl=3.7037037037037033
    http-/localhost-8080/thesis/zipcodes.wsdl=3.414634146341463
    http-/localhost-8080/thesis/currency.wsdl=2.631578947368421
    http-/localhost-8080/thesis/Weather.wsdl=0.9900990099009901
    http-/localhost-8080/thesis/addresslookup.wsdl=0.9181636726546906
    http-/localhost-8080/thesis/USHolidayDates.wsdl=0.24096385542168677
    http-/localhost-8080/thesis/Dictionary.wsdl=0.0
```

Fig. 12. Matching with additional none annotated services



```
matches:
     http-/localhost-8080/thesis/HotSpotFinder.wsdl=50.0
     http-/localhost-8080/thesis/AccountDetail.wsdl=40.0
     http-/localhost-8080/thesis/Orders.wsdl=25.0
     http-/localhost-8080/thesis/GoogleMapper.wsdl=20.0
     http-/localhost-8080/thesis/DictionaryOWL.wsdl=14.285714285714285
     http-/localhost-8080/thesis/OrderDetail.wsdl=14.285714285714285
     http-/localhost-8080/thesis/CustomerDetail.wsdl=12.5
     http-/localhost-8080/thesis/TrackingDetail.wsdl=12.5
     http-/localhost-8080/thesis/SourceInterface.wsdl=7.142857142857142
     http-/localhost-8080/thesis/SourceInterfaceV1.wsdl=7.142857142857142
     http-/localhost-8080/thesis/TargetInterface.wsdl=5.0
     http-/localhost-8080/thesis/TargetInterfaceV1.wsdl=5.0
     http-/localhost-8080/thesis/getPrimeNo.wsdl=4.545454545454546
     http-/localhost-8080/thesis/isbn.wsdl=4.545454545454546
     http-/localhost-8080/thesis/bnprice.wsdl=4.545454545454546
     http-/localhost-8080/thesis/numberconversion.wsdl=4.3478260869565215
     http-/localhost-8080/thesis/braille.wsdl=3.7037037037037033
     http-/localhost-8080/thesis/zipcodes.wsdl=3.414634146341463
     http-/localhost-8080/thesis/currency.wsdl=2.631578947368421
     http-/localhost-8080/thesis/Weather.wsdl=0.9900990099009901
     http-/localhost-8080/thesis/addresslookup.wsdl=0.9181636726546906
     http-/localhost-8080/thesis/USHolidayDates.wsdl=0.24096385542168677
```

Fig. 13. Matching with annotated dictionary service

This is not surprising as these components describe functions unrelated to the applications being built in these experiments.

Experiment four further demonstrates that consistently annotated Web services will score higher in matches than randomly picked Web services without any annotation. It also shows that the semantic matching algorithm is scalable. The time it takes to perform the match is proportional to the number of Web services.

## 5 Related Work

Despite the fact that the mashup tools share a common goal—enabling users to create situational applications based on existing application components—the actual capabilities, implementation technology and the target audience of these tools are widely different. For example, IBM's DAMIA [26], MashupHub [27], SABRE [28] or Apatar [29] largely target enterprise intranet environments, whereas Popfly [30] or Intel Mash Maker [31] are aimed at individual users and private use. The execution environment of a mashup could be on a server, client (i.e., browser) or a stand-alone desktop application.

Therefore, to focus our discussion, we compare mashup tools and approaches with our work mainly from the following

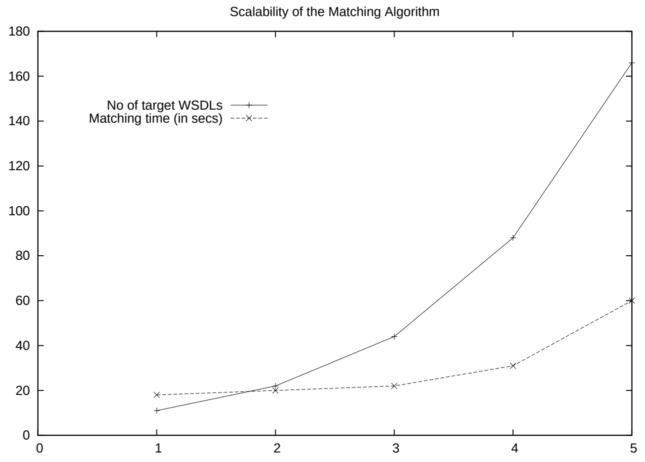


Fig. 14. Performance of the matching algorithm

three view points: i) types and extensibility of components, ii) support structure for users to find suitable components, and iii) alternative mashup programming patterns for wiring the components. As a number of research works suggest ([32], [28], [33], [34]), most tools provide limited search and discovery for mashup components. Users still need to know how to write code (e.g., JavaScript or XML/HTML) and link the components using technical concepts derived from programming. The following discussions will highlight that our work tries to overcome these issues through the use of composite applications running in Lotus Expeditor.

### 5.1 Types of Components in Tools

Yahoo! Pipes [35] provides a Web-based means of pulling data from various data sources, merging and filtering the content of those sources, transforming the content, and outputting the content for users to view or for use as input to other pipes. There are several limitations in Yahoo! Pipes. The first one is the limited set of inputs and outputs on components. There is no way to use arbitrary inputs or outputs when using this application. A component in a composite application should be able to accept many different types of inputs and provide many different types of outputs. The second limitation is that the flow of a pipe is static and sequential. While a user can configure many different inputs, all of the connections are executed in a sequential manner until the single output is reached. With our composite applications, the different components in the application can communicate with each other in any manner that the assembler chooses. Finally, Yahoo! Pipes is a server-based technology. There is no way for a user to construct and execute a pipe without a network connection and execute the pipe using locally stored data. A pipe can be accessed programmatically, like a Web service, but in order to execute the pipe the user must be able to connect to the Yahoo! Pipes server. Similar limitations exist in other Web portal type solutions such as Popfly [30] or Marmite [5].

DAMIA [26] extends the type of data sources for mash up to enterprise types such as Excel, Notes, Web services, and XML document rather than just URL based sources as in Yahoo! Pipes. It has a simple model of treating all data as sequences of XML. DAMIA offers three kinds of main operators, namely *ingestion*, *augmentation*, and *publication*. Ingestion brings data sources into the system. Augmentation provides extensibility to the system. It allows creation of new mashup operators and is thus more powerful than the fixed Yahoo! Pipes operators. Finally, publication operator transforms the output from a mashup to common output formats such as Atom, RSS or JSON for the consumption of other components. It relies on additional tools like QEDWiki[2] to visualize outputs. DAMIA focuses on data rather than component mashup. In contrast, we treat both data and applications as components.

## 5.2 Mashup Component Search and Discovery

As mentioned earlier, most mashup platforms have inappropriate support for component cataloging and querying. There are a few works that try to address this issue in different ways.

Many works use a Web 2.0 or online community-style approach. For example, Intel Mash Maker observes the user's behaviour (e.g., what kind of data she is interested in) and recommends an existing mashup that the user would find useful. It also correlates the user's behaviour with that of other users and use the knowledge to suggests mashups defined by other users on the same Web page. Most Web-based mashup tools offer a community feature where mashups are tagged, rated and organized by categories.

For data sources that publish the standard RDF, a tool such as Semantic Web Pipes [36], which is inspired by Yahoo! Pipes, offers a way to aggregate data using SPARQL [37] and RDF-aware operators. In Intel Mash Maker [31], much of the mashup creation and execution happens on the user's browser, directly over the Web pages currently on display. To extract data from Web pages, the tool uses an RDF schema associated with each page or a knowledge-base created by a community of users. Users can formulate XPath queries over the extracted schema to create intricate data model to manipulate with. A mashup is created by combining data from multiple pages in the form of mini-applications (or widgets). DAMIA also offers a mashup functionality for data sources with available RDF.

Other approaches focus on making "smart" guesses and recommendations for the users to choose suitable components for a given situation. A good example is MARIO (Mashup Automation with Runtime Orchestration and Invocation) [38]. It uses tag-based service description, service selection and taxonomy. The engine allows a user to explore the space of available mashups and preview composition results interactively, using tags, via an abstraction called "Wishful Search". MARIO offers a light weight planner that works with user generated tags for goal-driven based composition.

MatchUp [39] introduces the concept of auto-completion, very much like email addresses in a mail client or search phrases in a browser, to the creation of mashups. The idea is based on the observation that mashups developed by different users typically share common characteristics. The approach exploits these similarities to make ranked suggestions for possible "completions" (missing components and connections between them) for a partial mashup specification. These approaches share similar goals as ours in providing a rapid and end-user friendly composition framework via high-level semantic matching of available services, feeds and flows.

2. http://services.alphaworks.ibm.com/qedwiki, it is now part of IBM Lotus Mashups

## 5.3 Alternative Mashup Patterns

Conventional approach to mashup programming is to conceive mashup as data flow that takes input from multiple sources, applies transformation and visualizes the results. Normally the visual metaphor used in this environment is "boxes" (representing data sources) and "connectors/wires" (representing the flow). There are mashup tools that follow different programming patterns.

Karma [33] and UQBE [40] take a mashup as a schema matching or data integration problem. In this environment, disparate data sources are "joined" by common attributes as if joining relational tables. The proposed solution is based on a premise that it is easier for users to understand data semantics from concrete examples. Using a progressive approach to composing data (i.e., the Query By Example principles) is appealing to the non-programmers, and can be compared to our approach to suggesting semantically close components. However, the tools support data integration only and inherently dependent on domain specific characteristics of underlying data sources. It is not clear how a data source can be componentised and reused in a different situational application.

Recently, utilizing spreadsheet (tabular/grid) programming paradigms in data mashup is suggested. Mashroom [41] adopts nested relational model as underlying data model to represent Web-extracted data. A set of mashup operations is defined over the nested tables (e.g., merging, invoke and link another service directly on a range of rows in an iterative manner).

Another stream of work that is worth noting is mashup programming patterns at presentation level. That is, application/component integration is achieved purely through components that expose user interfaces only. Here, component models specify characteristics and behaviors of presentation components and propose an event-based composition model to specify the composition logic (e.g., MixUp [42]). The focus of MixUp is on integration of applications at presentation level. They do not deal with semantic annotation of component and finding compatible components.

In Smashup (Semantic mashup) [11], the components to be mashed up is restricted to RESTful Web services that is semantically annotated using SA-REST. The role of SA-REST in Smashup is to enable automatic data mediation. Smashup editor provides an interface where a user can enter the URLs of the annotated RESTful Web services that need to be mashed up. Then the user needs to wire the appropriate inputs and outputs of the selected services. Once the complete service chain is specified, the user runs a command and the Smashup editor will generate an HTML form that represents the mashed up application. The process is similar to our mashup tool. However, our tool is not restricted to matching up of browser-based services. Our components can be as diverse as a GUI component, a widget, a server-side EJB component, a .NET component, or an Adobe FLASH. We leverage SAWSDL for the purpose of discovery and matching of services, not for data mediation. A RESTful Web service without annotation can be used in our framework just like any other type of components. For a RESTful Web service with annotation, its SA-REST has to be converted to SAWSDL before it can be used in our framework.

Finally, Kepler [43] is an open source scientific workflow system which allows scientists to compose a composite application (a.k.a workflow) based on available actors. An actor can be built from any kind of applications. However, Kepler is not based on SOA architecture and it requires very skillful low level Java programming to convert applications into actors which can be composed within Kepler framework. Although in our current implementation, we do not provide tools for developers to convert existing components into annotated components that can be used in our mashup tool, our framework support SOA standard and exposing components' input and output as WSDLs with semantic annotation is a small effort compared to actors programming in Kepler.

## 6 Conclusion

One of the most difficult problems faced by users in a rich client environment is finding compatible and complementary components in a large catalog of components that have been built by different groups, at different times, using different technologies and programming conventions, as well as reusing those components as it is in a different application. In this paper, we have demonstrated that this problem can be largely solved by applying technologies related to the semantic web and Web services matching and using a progressive composition framework. The first technology that can be applied is Semantic Annotations for WSDL (SAWSDL), as standardized by the W3C. By adding model references to the message elements of the WSDL, the properties exposed by the component can be better described using modeling languages. Since the semantic modeling attributes can be added to any elements of the WSDL, the definition of the component could be further refined and described via annotations. Similarly, non-functional description of components can be added by introducing additional elements in the WSDL file. One limitation with using SAWSDL or any other annotation techniques is that component must be annotated a-priori. If a particular capability of a component is not being annotated, at runtime, it is impossible to leverage that capability for mashup even if it is useful to utilize that capability within a composite application. Part of our future work includes allowing components to dynamically expose their capabilities for mashup.

The second technology group that can be applied is the searching and matching algorithms created for use with Web services. These algorithms provide a powerful method for scoring the compatibility of an application component from a large set of possible component choices based on component capabilities. This scoring approach simplifies the application creation process for the composite application assembler by providing a ranking of potential components. This allows the assembler to focus on the highest ranked components, skipping over the lower ranked components, when considering which items may be compatible in the application being created. The searching process is further improved based on the fact that a composite application can be viewed and described as a single component when searching against a repository of components. This is done by creating a merged WSDL from each of the component of the composite application.

As demonstrated in the experiment results, the use of individual matching may still be valuable, especially when attempting to distinguish between components that score very closely to each other. A potential improvement to the analysis results would be to display the score for each target component using both the merged matching and individual matching, when the collection of scores is relatively close. In addition, both functional and non-functional descriptions are needed in order to make the matching more valuable for users. We are currently working on 1) allowing a user to specify only a specific set of inputs or outputs to consider during matching; 2) allowing a user to specify a weight on certain sets of inputs or outputs that affect the overall score of the matching.

One advantage of our composition framework is that end users do not need to concern low-level control-flow constructs during composition. This may be fine with simple application that involves a few components. However, in order to compose complex application that are robust, some forms of control-flow is necessary. Thus, a larger direction of future work is combining a service mashup approach with a process based integration approach. A semantically rich process language with constructs for conditionals, iterations and methods for insuring reliability of an integrated application can facilitate more complex combination of a larger set of applications. It can also help the analysis of an integration specification for general properties such as lack of a deadlock or a cycle and problem domain specific properties such as compliance of an integrated application to a set of business rules.

## References

[1] "Business Process Execution Language for Web Services version 1.1," http://www.ibm.com/developerworks/library/specification/ws-bpel/, 2007.

[2] "OSGi -The Dynamic Module System for Java," http://www.osgi.org, 2008.

[3] "Google Code Search," http://www.google.com/codesearch, 2008.

[4] J. Yu, B. Benatallah, F. Casati, and F. Daniel, "Understanding Mashup Development," *IEEE Internet Computing*, vol. 12, no. 5, pp. 44–52, September–October 2008.

[5] J. Wong and J. I. Hong, "Making Mashups with Marmite: Towards End-user Programming for the Web," in *Proc. of the SIGCHI Conf. on Human Factors in Computing Systems (CHI'07)*, New York, NY, USA, 2007, pp. 1435–1444.

[6] "Semantic Annotations for WSDL and XML Schema," http://www.w3.org/TR/sawsdl, 2007.

[7] "JSR 168: Portlet Specification," http://www.jcp.org/en/jsr/detail?id=168, 2008.

[8] "IBM Lotus Expeditor," http://www.ibm.com/software/lotus/products/expeditor, 2007.

[9] "Find Wi-Fi Hotspots Worldwide," http://www.jiwire.com, January 2009.

[10] "A Collaborative Environment for Creating and Executing Scientific Workflows," http://Kepler-project.org/, 2008.

[11] A. P. Sheth, K. Gomadam, and J. Lathem, "SA-REST: Semantically Interoperable and Easier-to-Use Services and Mashups," *IEEE Internet Computing*, vol. 11, no. 6, pp. 84–87, 2007.

[12] J. D. Lathem, "SA-REST: Adding Semantics to REST-based Web Services," 2005, Master Thesis, University of Georgia, Athens, Georgia.

[13] H. Knublauch, R. Fergerson, N. Noy, and M. Musen, “The Protege-OWL Plugin: An Open Development Environment for Semantic Web Applications,” http://protege.stanford.edu/plugins/owl/publications/ISWC2004-protege-owl.pdf, 2004.
[14] S.-M. T. Shah, R. Akkiraju, R. Ivan, and R. Goodwin, “Searching Service Repositories by Combining Semantic and Ontological Matching,” in *Proc. of the Third Intl. Conf. on Web Services (ICWS’05)*, 2005.
[15] G. Miller, “Wordnet: A Lexical Database for English Language,” *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, November 1995.
[16] J. Lee, R. Goodwin, R. Akkiraju, Y. Ye, and P. Doshi, “IBM Ontology Management System,” http://www.alphaworks.ibm.com/tech/snobase, 2008.
[17] J. Koehler and B. Srivastava, “Web Service Composition: Current Solutions and Open Problems,” in *ICAPS ’03 Workshop on Planning for Web Services*, June 2003.
[18] B. Medjahed and A. Bouguettaya, “A Multilevel Composability Model for Semantic Web Services,” *IEEE Trans. of Knowledge and Data Engineering (TKDE)*, vol. 17, no. 7, pp. 954–968, 2005.
[19] Q. Sheng, B. Benatallah, Z. Maamar, and A. Ngu, “Configurable Composition and Adaptive Provisioning of Web Services,” *IEEE Trans. on Services Computing*, vol. 2, no. 1, pp. 34–49, January-March 2009.
[20] L. Zeng, A. Ngu, R. Benatallah, B.and Podorozhny, and H. Lei, “Dynamic Composition and Optimization of Web Services,” *Journal on Distributed and Parallel Databases (Springer-Verlag)*, vol. 24, no. 1-3, pp. 45–72, December 2008.
[21] S. McIlraith, T. Son, and H. Zeng, “Semantic Web services,” *IEEE Intelligent Systems*, vol. 16, no. 2, pp. 46–53, March/April 2001.
[22] A. Marconi, M. Pistore, and P. Traverso, “Automated Composition of Web Services: the ASTRO Approach,” *IEEE Data Engineering Bulletin*, vol. 31, no. 3, September 2008.
[23] G. Phifer, “Portals Provide a Fast Track to SOA,” *Business Integration Journal*, vol. 20, no. 4, Nov-Dec 2005.
[24] “XMethods Web Services Portal,” http://www.xmethods.net/, 2009.
[25] “The QWS DataSet,” http://www.uoguelph.ca/~qmahmoud/qws/index.html, 2009.
[26] E. Simmen, M. Altinel, S. Padmanabhan, and A. Singh, “Damia: Data Mashups for Intranet Applications,” in *Proc. of the 2008 ACM SIGMOD Intl. Conf. on Management of Data*, Vancourver, Canada, June 2008, pp. 1171–1182.
[27] “IBM MashupHub,” http://www-01.ibm.com/software/info/mashup-center/, 2009.
[28] Z. Maraikar, A. Lazovik, and F. Arbab, “Building Mashups for the Enterprise with SABRE,” in *Proc. of the Sixth Intl. Conf. on Service Oriented Computing (ICSOC’08)*, Sydney, Australia, 2008, pp. 70–83.
[29] “Apatar,” http://www.apatar.com/, 2008.
[30] “Microsoft Popfly,” http://www.popfly.net/, 2007.
[31] “Intel Mash Maker,” http://softwarecommunity.intel.com/articles/eng/1461.htm, 2007.
[32] G. D. Lorenzo, H. Hacid, H. young Paik, and B. Benatallah, “Data Integration in Mashups,” *SIGMOD Record*, vol. 38, no. 1, pp. 59–66, 2009.
[33] R. Tuchinda, P. Szekely, and C. A. Knoblock, “Building Mashups By Example,” in *Proc. of Intl. Conf. on Intelligent User Interfaces*, Gran Canaria, Spain, 2008, pp. 139–148.
[34] V. Hoyer and M. Fischer, “Market Overview of Enterprise Mashup Tools,” in *Proc. of the Sixth Intl. Conf. on Service Oriented Computing*, Sydney, Australia, 2008, pp. 708–721.
[35] “Yahoo Pipes,” http://pipes.yahoo.com/pipes/, 2007.
[36] D. Le-Phuoc, A. Polleres, M. Hauswirth, G. Tummarello, and C. Morbidoni, “Rapid Prototyping of Semantic Mash-ups Through Semantic Web Pipes,” in *Proc. of the 18th Intl. World Wide Web Conf. (WWW’09)*, New York, NY, USA, 2009, pp. 581–590.
[37] SPARQL Query Language for RDF. [Online]. Available: http://www.w3.org/TR/rdf-sparql-query/
[38] A. Riabov, E. Bouillet, M. Feblowitz, Z. Liu, and A. Ranganathan, “Wishful Search: Interactive Composition of Data Mashups,” in *Proc. of 17th Intl. World Wide Web Conf. (WWW’08)*, Beijing, China, April 2008, pp. 775–784.
[39] O. Greenshpan, T. Milo, and N. Polyzotis, “Autocompletion for Mashups,” in *Proc. of the 35th Intl. Conf. on Very Large Data Base (VLDB’09)*, Lyon, France, 2009, pp. 538–549.
[40] J. Tatemura, S. Chen, F. Liao, O. Po, K. S. candan, and D. Agrawal, “UQBE: Uncertain Query By Example for Web Service Mashup,” in *Proc. of the 2008 ACM SIGMOD Intl. Conf. on Management of Data*, Vancourver, Canada, 2008, pp. 1275–1279.
[41] G. Wang, S. Yang, and Y. Han, “Mashroom: End-User Mashup Programming Using Nested Tables,” in *Proc. of the 18th Intl. Conf. on World Wide Web (WWW’09)*, New York, NY, USA, 2009, pp. 861–870.
[42] J. Yu, B. Benatallah, R. Saint-Paul, F. Casati, F. Daniel, and M. Matera, “A Framework for Rapid Integration of Presentation Components,” in *Proc. of the 16th Intl. World Wide Web Conf. (WWW’07)*, New York, NY, USA, 2007, pp. 923–932.
[43] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, “Scientific Workflow Management and the Kepler System,” *Concurrency and Computation: Practice & Experience*, vol. 18, no. 10, pp. 1039–1065, 2006.

**Anne H.H. Ngu** is currently an Associate Professor with the Department of Computer Science at Texas State University-San Marcos. From 1992-2000, she worked as a Senior Lecturer in the School of Computer Science and Engineering, University of New South Wales (UNSW). She has held research scientist positions with Telecordia Technologies and Microelectonics and Computer Technology (MCC). She was a summer faculty scholar at Lawrence Livermore National Laboratory from 2003-2006. Her main research interests are in information integration, service oriented computing, scientific workflows and agent technologies.

**Michael P. Carlson** is a Senior Software Engineer at IBM, the lead developer for the Lotus Expeditor Client, and the architect of the Lotus Expeditor Toolkit. He has been working with Eclipse technology and OSGi technology since before they worked with each other. Carlson has been at IBM since 1998 and has worked on a variety of products including printers, operating systems, network appliances, web applications, software development tools, devices, and desktop runtime environments.

**Quan Z. Sheng** is a senior lecturer in the School of Computer Science at the University of Adelaide. His research interests include service-oriented architectures, distributed computing, and pervasive computing. He is the recipient of Microsoft Research Fellowship in 2003. He is the author of more than 70 publications. He received a PhD in computer science from the University of New South Wales, Sydney, Australia. He is a member of the IEEE and the ACM.

**Hye-young Paik** is a lecturer at the School of Computer Science and Engineering in University of New South Wales. Her research interests include flexible business process modelling, modelling and reuse issues in Web service mashups. She is an active member of Web services research community and publishes in international journals and conferences regularly. She received her PhD in computer science from University of New South Wales, Sydney, Australia.