Automatic Mash Up of Composite Applications

Michael Pierre Carlson[1], Anne H.H. Ngu[1], Rodion Podorozhny[1], and Liangzhao Zeng[2]

[1] Computer Science Dept., Texas State University, San Marcos, TX 78666, USA
{mc1173,rp31,hn12}@txstate.edu
[2] IBM T.J. Wastson Research Center, Yorktown Heights, NY 10598
lzeng@us.ibm.com

Abstract. The need for integration of both 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 become a popular platform in the IT industry for building such composite applications recently with the integrated components being provided as web services. A key limitation of such a web service 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 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 experimental study on the above progressive composition framework on IBM's Lotus Expeditor which is an extension of a Service Oriented Architecture (SOA) platform called the Eclipse Rich Client Platform (RCP) that complies with the OSGi standard.

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 the 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] or UML statechart [2]. The web services 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 OSGi [3], 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 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 mash up 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 that is returned is typically based on criteria that have no relevance to the application assembler (e.g. alphabetical or last update time). Interfaces like Google Code Search [4] 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 the user in locating compatible components for 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 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 difficult process up front, or could result in repeated cycles of trial and error, especially when the target environment supports a variety of technologies.

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. End users, at least the savvier end users, should be able to compose applications with minimal training. For a call center in an enterprise, this may simply mean being able to assemble a composite application on the fly that takes a caller's information in one application and has the input reflected in other applications that are currently running on his or her desktop for other contextual information. For the savvy end user at home or 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.

The main contributions of this paper are as follows. First it is shown 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 rich client composite applications. These components may include graphical user interfaces, which are not an artifact in web service components. By building on the techniques initially developed for web services matching, the problems associated with finding useful and valid components for composite applications using high level concepts is possible. This enables the progressive construction of composite applications from a catalog of available components without deep knowledge of the components in the catalog. We demonstrated how the additional characteristics of components, specifically graphical user interface details, can be modeled, described using Semantic Annotation for web services (SAWSDL) [5] and matched in a similar fashion to the programmatic inputs 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 much better results for the user when searching for components to integrate into the composite application. Finally, it will be shown using sample applications and scenarios that by taking into account the unique characteristics of a component (i.e. coexistence of user interface components), and new techniques of merging semantic descriptions across multiple components, we can achieve a much more accurate search result for compatible components.

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 user interfaces, built from technologies such as JFace, JSPs, HTML, Eclipse Standard Widget Toolkit (SWT), Swing, native windowing systems, etc. Like web services and Enterprise Java Beans, application components can take programmatic inputs and provide programmatic outputs. In application components, 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 is a Portlet [6].

We adopt the IBM Lotus Expeditor [7] platform to develop application components and composite applications. Expeditor contains a Composite Application Infrastructure (CAI) and an associated Composite Application Editor (CAE). 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 application components of a composite application. 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. 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 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, portal server, or Lotus Domino NSF database.

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 when the components are installed or imported into Lotus Expeditor. In the current implementation, the WSDL files for application components in CAI does not include the graphical user interface type (e.g. JSP, SWT, 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 deployment platform does not provide support for Portlet interfaces, then 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 did not provide a way for finding compatible and complementary components from a catalog of existing components based on components' capabilities. We extended the Expeditor Workbench with a new menu item called Analyze Composite App. which will open a dialog box for user to search for the desired components to use for composition. The following section illustrates in Figure 1 the sequence of screen shots in Lotus Expeditor Client workbench that resulted in a simple HotSpotFinder composite application.

2.2 A Scenario of Developing Composite Application

Screen A shows the initial composition workbench. On the right is the panel that shows the list of components (e.g. HotSpotFinder, GoogleMapper, CityStatePicker, OrderTracker) that are available for composition as well as links to available remote components. On the left is the panel that displays all the existing composite applications of a particular user (there is only one composite application available beside the one that is being composed). The middle panel displays the in-progress composite application. When the user clicks on Analyze Composite App. menu, a dialog box in screen B is displayed. After the user entered the desired search criteria at the top of screen and pressed the "Find Matches" button, screen C is displayed. By picking the best matched component (the one with the highest score) from the palette and dropped 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 CityState Picker 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

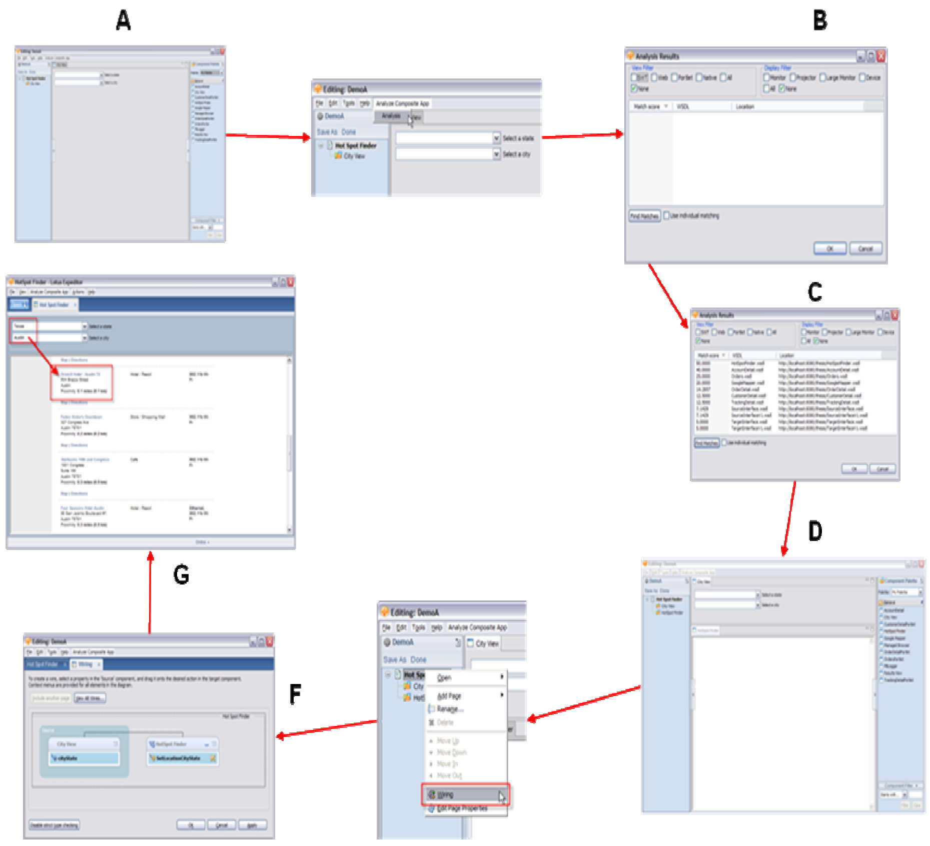


Fig. 1. Sequence of steps in composing an application

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.

3 Composite Application Matching

In this section, we demonstrate techniques, technologies, and algorithms that can be leveraged to provide support for finding compatible components. The WSDL-based programmatic inputs and outputs of a component in CAI can be further modeled using semantic web techniques to enable searching using higher level concepts. Additionally, the implementation technology (e.g. SWT, Swing, etc) used to create the graphical user interface can be modeled using semantic model and added to component's WSDL to further describe the platforms that the component can support.

The additional metadata added did not change the nature 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 is described by T. Syeda-Mahmood [8]. This work describes combining the use of semantic and ontological matching for the purposes of matching web service components. In fact, the matching code that was used to produce the results shown by T. Syeda-Mahmood is the same matching code that is used to conduct the experiments in Section 4. In brief, the matching algorithm works as follows:

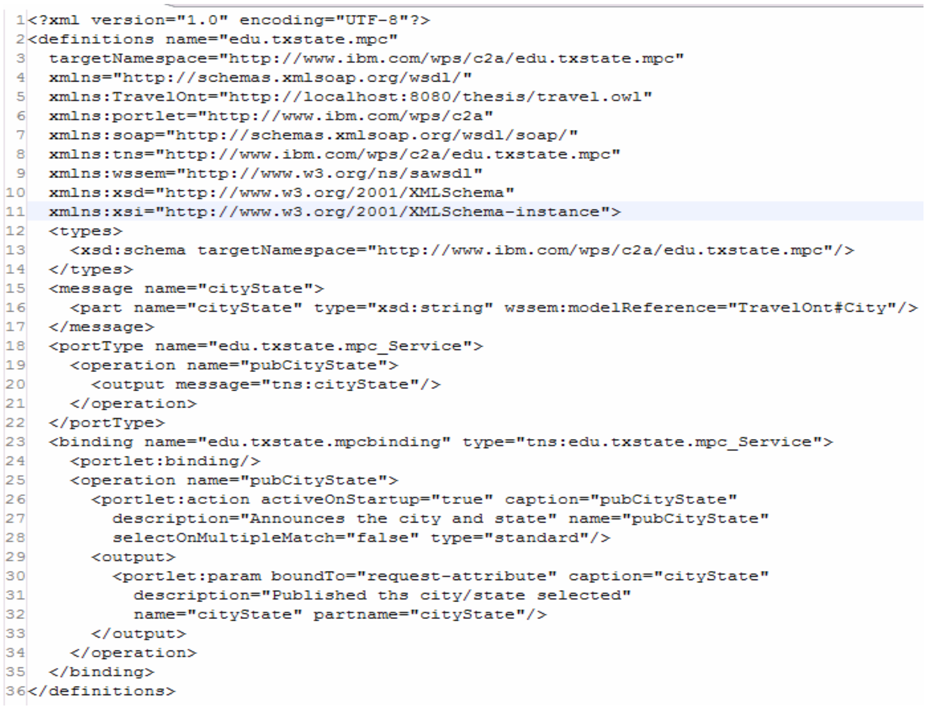
1. Using a single input WSDL and a collection of target WSDLs, a score is calculated based on the number of matching terms found between the input WSDL and each target WSDL. A thesaurus, in this case WordNet [9], is used to expand the matching to include synonyms. Thus, this phase is focused on keyword matching.
2. A second search is then invoked using the same input and targets as above. This time the semantic annotations specified in the WSDL files are considered. The semantic models for each component are compared using a custom ontology matching algorithm. This algorithm takes into account the relationships between the elements given, such as inheritance, hasPart, hasProperty, etc. A score for each combination is calculated based on the number of attributes that are matched for each combination.
3. The final score is calculated using a winner-takes-all approach. The maximum of the first score and the second score is reported as the overall matching score for each input and target combination.

Even though we can reuse much of the same logic and algorithms, there are a few fundamental differences when dealing with non-web services based components. In many cases, when searching for a web service, the developer is looking for APIs that can either:

1. Match – Using the output from a single web service and finding a second web service that can take that as input. The developer can continue this process and string together several web services with application specific control-flow pattern in order to complete a business process.
2. Compose – Starting with a known output and a known input, the developer uses search techniques that can allow them to find 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.

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 with specific control flows; rather, the goal is to integrate separately created components together "on the glass" [10] and provide the ability for those applications to communicate or interact without prior knowledge of each other or in any specific order. This means that a composite application may bring together a human resources vacation planning component with a project management component. By linking the two applications together, the project management component could potentially use vacation data in the vacation planning component to adjust project schedules. In no way, however, does the process of scheduling vacation need to be modified in order to use the data. Additionally, the developer of each of these components does not need to have knowledge of the implementation of the other component or how to invoke the programmatic interfaces.

The markup required for composite application matching is similar to the markup required for web services, at least with respect to data inputs and outputs. Figure 2. shows an example of a CityStatePicker component with semantic annotation. A CityStatePicker component allows a user to select a valid city and a state from given lists.



```
 1<?xml version="1.0" encoding="UTF-8"?>
 2<definitions name="edu.txstate.mpc"
 3  targetNamespace="http://www.ibm.com/wps/c2a/edu.txstate.mpc"
 4  xmlns="http://schemas.xmlsoap.org/wsdl/"
 5  xmlns:TravelOnt="http://localhost:8080/thesis/travel.owl"
 6  xmlns:portlet="http://www.ibm.com/wps/c2a"
 7  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 8  xmlns:tns="http://www.ibm.com/wps/c2a/edu.txstate.mpc"
 9  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 ths city/state selected"
32          name="cityState" partname="cityState"/>
33      </output>
34    </operation>
35  </binding>
36</definitions>
```

Fig. 2. WSDL with semantic markup for CityState Picker 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 user interacts with this component. On line 16 you will see that this message has been annotated with a reference to an element in a semantic model. This is shown as wssem:modelReference="TravelOnt#City" in the WSDL file. TravelOnt is an OWL-based semantic model that is available on the web. With this annotation, we are describing the message in terms of an OWL class in an OWL model. Continuing along the web services methodology, 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). 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 web service matching technologies. By including the annotation, the matching engine is able to match based on capabilities of the component as described in the OWL model. For example,

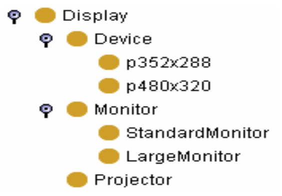


Fig. 3. OWL model for display

assume the "cityState" component were to be compared against another component that contained the 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 "Trave-lOnt#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 match would score higher because the analysis would show that these two elements are very closely related.

Semantic annotations only work if there is a unified ontology model. Therefore, it may be necessary to create a semantic model for the components if none exists. We use Protégé-OWL [11] editor to create OWL-based semantic models for describing application component's GUI characteristics since none exists. Figure 3 shows the simple GUI semantic model used in our framework. This model describes a single top-level OWL class named Display. There are three subclasses of Display, namely Device, Monitor, and Projector. Further, there are two subclasses of Device, p352x288 and p480x320. The two resolutions for the device class describe certain types of device interfaces. The p352x288 represents many Windows Mobile smart phones and the p480x320 represents Apple's first generation iPhone. The Monitor class is further subclassed into StandardMonitor and LargeMonitor.

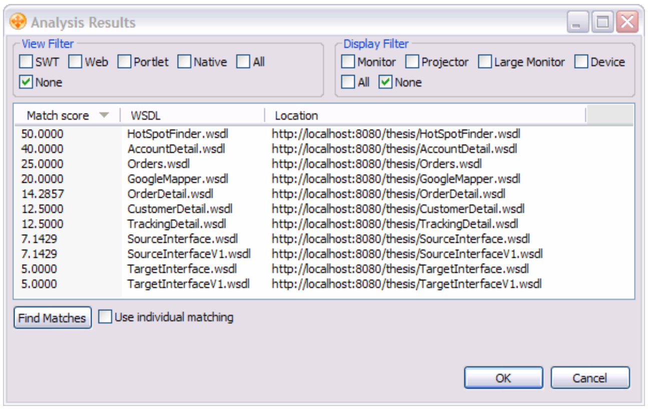
4 Experimental Study

To validate the suggested automatic mashup approach, several experiments were performed. These experiments involved using 12 components created with the help of the Java Eclipse IDE toolkit. Once created, they were installed in Lotus Expeditor. The semantic matching algorithm to be validated was implemented as part of the Analyze Composite Application (App.) in Lotus Expeditor workbench. A Lenovo ThinkPad T60p running Microsoft WindowsXP SP2 served as an experimentation platform. In the experimental test bed, the Analyze Composite App first reads the content of the current composite application. The "View Filter" and "Display Filter" sections in Analyze Composite App. enable the user to set the graphical user interface search criteria. The main table displays the score associated with each of the matching target WSDLs (i.e. component descriptions) that were in the repository. The "Find Matches" button starts the search process. 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. If not checked, a merged WSDL across all existing components in the current composite application will be used in the matching process. We validated the capabilities of the approach by running five experiments on two different composite applications. The first composite application that we studied is hotspot tracking and the second application is an order tracking.. Similar results is obtained from the order tracking application and thus we only report three experiments that are related to hotspot tracking application. The first experiment is to validate the fact that the semantic matching algorithm together with SAWSDL annotation indeed provides a better match result than matching without using the semantic annotation. The second experiment is to show increased accuracy in the search process when using a merged WSDL. Finally, the last experiment shows the effect of adding the GUI semantic information for the matching process.

The first experiment involves simple matching using two component WSDLs and the semantic web matching logic described in Section 3. The input for this matching scenario is CityStatePicker component that allows a user to first select a state from a select box and then select a city from a second select box. After the city is selected, the component publishes the selected city and state information. The target component is the HotSpotFinder component. This is an Eclipse SWT Browser which accesses JiWire website for a listing of wireless internet access points in the given city and state. When semantic web matching is applied to CityStatePicker component, a score of 50 for HotSpotFinder is produced. In order to show the effect of the semantic matching only, a modified version of the HotSpotFinder.wsdl is used and the same experiment is run again. In the modified version, the identifying names such as city and address are replaced with random strings. The resultant score which uses solely the ontological match is 37.50 for HotSpotFinder. This lower score is likely due to the fact that only the message elements in the WSDL file have semantic models attached to them. 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 other 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 in this experiment and that we have a valid environment for conducting other experiments.

The second experiment shows the effect of using the merged WSDL search request to find compatible components for a composite application. In this case, the target WSDLs will be chosen from the complete collection of components available in our framework. In order to start the composition process, we must have a starting component. The CityStatePicker component is used as the first component. This composite application is then matched against the complete catalog of components. The results, as shown in Figure 4, tell us that the HotSpotFinder component has the highest score (50) and should be added to the application.

Once the HotSpotFinder component is added to the application, the analysis is run again using a merged WSDL search request from both CityStatePicker and HotSpotFinder. This time the highest ranking component returned is the GoogleMapper component. This component takes as input an address, and based on this address, the component loads a map for the address using Google Map to provide the actual content. In fact, not only is this component now the highest scoring component, but it has


Analysis Results
View Filter
☐ SWT ☐ Web ☐ Portlet ☐ Native ☐ All
☑ None
Display Filter
☐ Monitor ☐ Projector ☐ Large Monitor ☐ Device
☐ All ☑ None
Find Matches
☐ Use individual matching
OK
Cancel

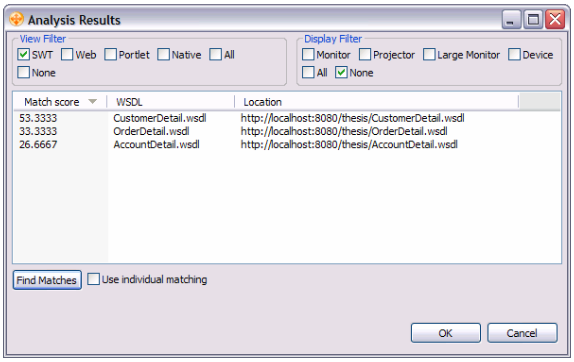
Match score	WSDL	Location
50.0000	HotSpotFinder.wsdl	http://localhost:8080/thesis/HotSpotFinder.wsdl
40.0000	AccountDetail.wsdl	http://localhost:8080/thesis/AccountDetail.wsdl
25.0000	Orders.wsdl	http://localhost:8080/thesis/Orders.wsdl
20.0000	GoogleMapper.wsdl	http://localhost:8080/thesis/GoogleMapper.wsdl
14.2857	OrderDetail.wsdl	http://localhost:8080/thesis/OrderDetail.wsdl
12.5000	CustomerDetail.wsdl	http://localhost:8080/thesis/CustomerDetail.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

Fig. 4. Matching of CityStatePicker component

also shown a substantial jump from its previous score of 20 to the new score of 50. This result indicates that the GoogleMapper component is a good candidate to add to the application at this time. We further validated the effect of matching with a merged WSDL request in Order Tracking composite application which won't be shown here due to lack of space.

The third experiment showed the effects of using GUI information for conducting the match. We used the OrderTracking composite application to illustrate that. It is done so because this application can be composed from components with greater variety of interface technologies. In order to conduct this experiment, a new message type pertinent to interface technology will be added to the search component WSDL and each of the target components' WSDLs. For this experiment, the Customer Details component is marked as having an SWT GUI, the Order Details component is marked as having a Portlet GUI, and the Account Details is marked as having a Web Browser GUI. The test scenario is the same as in experiment two with the merged WSDL search. Initially, the Orders and Order Tracking components are selected for the composite application and a merged WSDL search is executed. By selecting one or more of the checkboxes in Analyze Composite App. search dialog for SWT, Web, Portlet, Native, or All, the merged search WSDL will be enhanced with this additional criterion. In the first run we will select SWT checkbox. The match score for the Customer Detail component has increased slightly and the scores for the other components have decreased. Customer detail was originally the best match and it remains so with this additional filter. Components that do not have GUI semantic annotation won't even appear as part of the result. This is shown in Figure 5.

Next, let us summarize the results of the experiments described above. Experiment one has shown that the function provided by 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 function can be used without adding the semantic markup. However, as we saw 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.



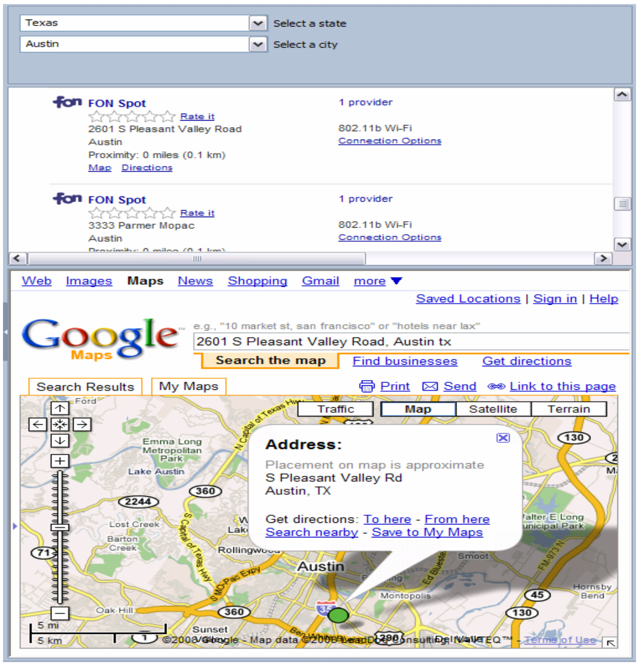
View Filter: [✓ SWT] Web Portlet Native All None

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

Match score	WSDL	Location
53.3333	CustomerDetail.wsdl	http://localhost:8080/thesis/CustomerDetail.wsdl
33.3333	OrderDetail.wsdl	http://localhost:8080/thesis/OrderDetail.wsdl
26.6667	AccountDetail.wsdl	http://localhost:8080/thesis/AccountDetail.wsdl

Use individual matching

Fig. 5. Matching of components with GUI semantic annotation



Texas Select a state
Austin Select a city

FON Spot
Rate it
2601 S Pleasant Valley Road
Austin
Proximity: 0 miles (0.1 km)
Map Directions
1 provider
802.11b Wi-Fi
Connection Options

FON Spot
Rate it
3333 Parmer Mopac
Austin
1 provider
802.11b Wi-Fi
Connection Options

e.g., "10 market st, san francisco" or "hotels near lax"
2601 S Pleasant Valley Road, Austin tx

Address:
Placement on map is approximate
S Pleasant Valley Rd
Austin, TX

Get directions: To here - From here
Search nearby - Save to My Maps

Fig. 6. Completed HotSpotFinder application

Experiment two has shown that it is possible to build a composite application from a collection of different components 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. As the HotSoptFinder application was composed, the initial results did not show as much difference in scores as might be expected, but with the addition of the HotSpotFinder component, we can then construct a merged

WSDL search request. This enables the GoogleMapper component to stand out as the next obvious addition.

Experiment three illustrated that the score of a component can be impacted by the addition of the GUI semantics. It can therefore be summarized that any potential metadata could be added to the WSDL and used in the semantic matching logic later on.

Thus, the experiments enabled us to validate the semantic service matching approach. The tool based on the approach delivered reasonable matches for service specifications thus providing appropriate functional service compositions. The picture of a completed integrated application (HotSpotFinder Application) is shown in Figure 6.

5 Related Work

There are several other similar approaches to automatic mashing of application compositions. Two of those include Yahoo! Pipes [12] and the various "Mashup" products, such as Intel Mash Maker [13] and Mash-o-matic [14]. While interesting in their own respects, there are limitations in these technologies that are solved through the use of composite applications running in Lotus Expeditor.

Yahoo! Pipes provides a web-based means of pulling data from various data sources, merging and filtering the content of those sources, transforming the content, and finally outputting the content for users to view or for use as input to other pipes. There are several limitations in Yahoo! Pipes. The first is the limited set of inputs and outputs. There is no way to use arbitrary inputs or outputs when using this application. There are a limited number of input types and output types from which the user can select. A component in a composite application should be able to accept many different types of inputs and provide many different types of outputs. Secondly, 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 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 that makes use of only a web user interface. 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. These same limitations exist in other web portal type solutions such as iGoogle and My Yahoo.

Damia [17] extends the type of data sources that can be used for mash up to enterprise types such as Excel, Notes, web services repository and XML rather than just URL based sources as in Yahoo pipes. It has a simple model of treating all data as sequences of XML. There are three kinds of main operators, ingestion, augmentation, and publication. Ingestion bring data sources into the system, Augmentation operator provides extensibility to the system. It allows creation of new mash up operators and is thus more powerful than the fixed Yahoo pipes operators. Finally, there are publication operators which transform the mashup to common output formats such as Atom, RSS or JSON for the consumption of other components. Damia is centered on data rather than component mashup.Another work with a somewhat similar purpose is the COmposer of Integrated Systems (COINS) by Mark Grechanik and Kevin Conroy

[15]. The COINS enables putting together GUI-based applications exploiting an accessibility programmable interface that manipulates the user interface components such as buttons, textfields, menu items and others. The use of the accessibility interface enables the COINS system to integrate applications that comply with the accessibility standard (section 508 of the rehabilitation act). Such a solution has lower performance than using a dedicated API, yet it significantly decreases the time needed to integrate applications (by an average of 3 times). The focus of the COINS system is on integration. They do not deal with semantic annotation of component and finding compatible components.

Kepler[16] 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. Kepler is also not meant to be used in a mobile rich client environment like Lotus Expeditor.

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 and reuses 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 like Lotus Expeditor. The first technology that can be applied is Semantic Annotations for WSDL (SAWSDL), as standardized by the W3C. By adding semantic model references to the message elements of the WSDL, the properties exposed by the component can be better described using modeling languages. Since the modeling attributes can be added to any elements of the WSDL, the definition of the component could be further refined and described using the concepts of SAWSDL.

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 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 seen 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 window 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.

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 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. `http://www.ibm.com/developerworks/library/specification/ws-bpel/`
2. `http://www.uml.org`
3. OSGi originally stood for Open Service Gateway Initiative. This term is no longer used and the alliance is now known simply as OSGi., `http://www.osgi.org`
4. `http://www.google.com/codesearch`
5. `http://www.w3.org/TR/sawsdl`
6. Portlets as implemented in the Java programming language are defined by JSR 168, `http://jcp.org/en/jsr/detail?id=168`
7. `http://www.ibm.com/software/lotus/products/expeditor`
8. Syeda-Mahmood, T.S., Akkiraju, R.I.A., Goodwin, R.: Searching Service Repositories by Combining Semantic and Ontological Matching. In: Third International Conference on Web Services, ICWS (2005)
9. Miller, G.: WordNet: A lexical database for the English language. Communications of the ACM (1983)
10. Phifer, G.: Portals Provide a Fast Track to SOA. Business Integration Journal (November/Deccenber 2005)
11. Knublauch, H., Fergerson, R., Noy, N., Musen, M.: The Protégé-OWL Plugin: An Open Development Environment for Semantic Web Applications (2004), `http://Protege.Stanford.edu/plugins/owl/publications/ISWC2004-protege-owl.pdf`
12. `http://pipes.yahoo.com/pipes/`
13. `http://softwarecommunity.intel.com/articles/eng/1461.htm`
14. Murthy, S., Maier, D., Delcambre, L.: Mash-o-matic. In: Proceedings of the 2006 ACM Symposium on Document Engineering, pp. 205–214 (2006)
15. Grechanik, M., Conroy, K.M.: Composing Integrated Systems Using GUI-Based Applications And Web Services. In: IEEE International Conference on Services Computing, SCC 2007 (2007)
16. Kepler Project: `http://Kepler-project.org/`
17. Simmen, E.D., Altinel, M., Padmanabhan, S., Singh, A.: Damia, data mashups for intranet applications. In: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, pp. 1171–1182 (2008)