RI 06009 22 November 2006 Computer Science

IBM Research Report

An Integrated Development Environment for Web Service Composition

Girish Chafle, Gautam Das, Koustuv Dasgupta, Arun Kumar, Sumit Mittal, Sougata Mukherjea, Biplav Srivastava
IBM Research Division
IBM India Research Lab
Block I, I.I.T. Campus, Hauz Khas
New Delhi - 110016. India.

IBM Research Division
Almaden - Austin - Beijing - Delhi - Haifa - T.J. Watson - Tokyo - Zurich

An Integrated Development Environment for Web Service Composition

Girish Chafle, Gautam Das, Koustuv Dasgupta, Arun Kumar, Sumit Mittal, Sougata Mukherjea, Biplav Srivastava

IBM India Research Laboratory
New Delhi, India

ABSTRACT

There has been considerable interest in the industry to adopt Service Oriented Architecture (SOA) and use Web services for encapsulating enterprise information assets. However, in order to realize the true potential of SOA, it is critical to develop the tools and technologies that enable dynamic composition of Web services to create new services with a richer set of functionalities. Unfortunately, little effort has been dedicated towards tooling for end-to-end service composition. What is required is an Integrated Development Environment (IDE) to ease the process of composition, thereby reducing development time and integration efforts. In this paper, we present an IDE for end-to-end web service composition. The IDE is based on a two–stage service composition approach – where suitable components are first discovered based on semantic annotations of existing services, and then a new service is stitched based on the developer's requirements. We present the design of the IDE, describe its integration with existing technologies, and discuss its usability based on the findings of a user survey.

1. INTRODUCTION

Web services encapsulate information, software, and other resources, and make them available over the network via standard interfaces and protocols. Web services provide an instantiation of the Service–Oriented Architecture (SOA) and facilitate the process of enterprise application integration [11, 15]. The SOA approach promises efficient automation of business processes by offering a standardized way to expose and access the functionalities of enterprise applications as (Web) services. It also provides an enterprise bus infrastructure for communication and management of these services. Finally, the SOA approach prescribes the use of BPEL (Business Process Execution Language) to develop applications quickly by defining the order in which services will be invoked. BPEL (or BPEL4WS), while being an effective means of specifying business processes and their interactions, cannot handle the task of creating new services dynamically (on-demand) from a repository of component services. In particular, when a functionality cannot be realized by existing services, some of the existing ones could be combined to provide the requested functionality. This process of *service composition* along with effective development environments to facilitate the quick and simple composition of Web services, remains a key challenge to realize the true potential of SOA.

From a research point of view, web services technology has reached a stage of considerable maturity. There have been many research efforts on web services and the problem of web service composition has been studied by a plethora of researchers [3, 12, 9, 17, 10]. In previous work, the authors studied the problem of web service composition, and identified the challenges in scalable service composition. The challenges range from problem characterization , to efficient information modeling [8], and handling of functional and non–functional requirements [2]. The authors addressed these issues in a principled web service composition approach [1] by introducing a differentiation between web service *types* and *instances*. The solution drives the composition process from specification, to creation of desired functionality using planning techniques on semantically annotated services, and finally selection of appropriate service instances that optimize non–functional requirements.

A number of tools have been made available by various vendors to create web services. However, little progess has been made in the direction of tooling for service composition. Most of the research projects on web service composition have led to the development of prototypes, e.g. E–flow [3], Self–Serv [12], METEOR–S [9], SHOP2 [17], SWORD [10] etc. Further, tools like *ontology browser, WSDL editor, BPEL editor* etc. have been made available by various sources and can be used by these prototypes. However, all these tools are piecemeal and do not provide an end-to-end view of service composition. What is required to facilitate web service composition is an Integrated Development Environment (IDE) to ease the process of composition, thereby reducing development time and integration efforts. The IDE should have an easy way to specify requirements of composite service in a precise manner. Further, it should be able to guide the user through the composition process, while allowing modifications or feedbacks at different stages.

Finally, the IDE should generate output in an industry standard language, like BPEL, that can be easily deployed.

With this in mind, we present the design and implementation of an IDE for composition of web services. One of the key features of the proposed IDE is that it strives to automate the core stages of the composition process, while leaving scope for valuable user feedback between the stages. To the best of our knowledge, no such IDE exists and this is the first effort in this direction. Our IDE is based on the two–stage service composition approach proposed in [1], which we refer to as the Synthy approach. To achieve the desired level of automation, the IDE uses Semantic Web technologies to describe component services and use them on demand. At the same time, we ensure that the environment is not burdened by the complexities of semantic technology, thereby making it simple to use by developers and business process analysts alike.

2. SYNTHY: A TWO STAGE SERVICE COMPOSITION APPROACH

We begin by describing the Synthy approach that enables semantic composition of Web services [1, 2]. The proposed approach introduces a differentiation between web service *types* and *instances*. Web service types are groupings of similar (in terms of functionality) services, while instances refer to the actual web services that can be invoked. A web service type is semantically described with the help of inputs, outputs, preconditions and effects (IOPE's) that capture the functionality offered by these type of services, with the IOPE parameters being defined with the help of a domain dependent ontology of terms. A web service instance, on the other hand, is described with the help of Web Service Description Language (WSDL) along with the non-functional (QoS) attributes associated with this instance. Our solution *Synthy* drives the composition process right from specification to final deployment and execution of a composite workflow. For this purpose, Synthy follows a staged approach where the functional composition, based on types, called the Logical Composition is decoupled from the non–functional composition based on instances, called the Physical Composition. Composition at the logical stage uses ontology matchmaking and planning techniques to generate an abstract workflow, consisting of service types, that meets the functional requirements of the composite service. Composition at the physical stage then binds each service type in the abstract workflow to a concrete service instance, while satisfying and optimizing the non-functional specifications of the composite service. At the end of this process, an executable workflow is generated that can be deployed and run on an execution infrastructure. Figure 1 illustrates the Synthy approach.

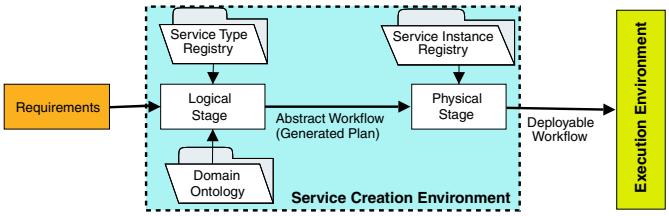

Figure 1: Synthy: A Two-stage Composition Approach

Sample Scenario Service providers, like telcos, are increasingly targeting enterprise customers because of the higher margins and longer-term relationships. Suppose a telco is attempting to automate a typical Helpline (or call center) for a washing machine manufacturer. In such a scenario, a customer calls in to report a problem with her washing machine. This problem needs to be assigned to an agent for resolution. If the problem is such that it could be solved over the phone, a desk-based agent at the call center will be assigned. Otherwise, a field agent who can visit the customer and fix her washing machine needs to be determined. The component services available in this scenario include Location tracking, SMS, Call Setup, Agent Expertise database, Problem Classification, Agent Selection etc. Some of these provide telco specific functions, e.g., delivering SMS text messages and location tracking of mobile phones. Others, like problem classification, are specific to the application domain. Finally, these components are exposed as Web services and can be combined to create the composite service.

For building the Helpline service, we do not wish to burden the developer with the arduous task of manually locating relevant services and connecting them up. Further, we wish to leverage semantic technologies that allows newly composed services to be annotated (meta–information about their functionalities) and reused in the future. The developer can compose the service in a bottom–up manner by composing first the sub–scenarios and then combining them. Suppose the developer decides to first build a Location-based Agent Selector (LAS) service. Given a customer's location and a list of agents out in the field, this service needs to select the agent closest in proximity to the customer's residence. The selected agent will then be asked to visit the customer and fix her washing machine. This LAS service, when properly annotated, can now be stored and reused in other compositions, such as the Helpline Service. In the rest of the paper, we use the LAS Service to drive the discussion on the proposed IDE and its effectiveness in composing Web services.

3. IDE FOR WEB SERVICE COMPOSITION

Most development environments enable software developers to manage the design-code-debug cycle. However, the activities involved are primarily offline ones and are not connected to the runtime infrastructure. The world of web services presents a different challenge, however. Unlike traditional software libraries that come in the shape of *off-the-shelf* components, web services are *actively running* components that need to be composed together. Moreover, new web services may come up or old ones may go down dynamically, leading to much more frequent changes than in traditional software libraries or components. Therefore, component updating mechanism based on the traditionally employed periodic software releases does not apply here. This means that a key characteristic of an IDE for web service composition is that it should be able to work with components in the runtime environment in addition to offline development modules. This has implications on

functionality, interface, performance and runtime behavior of the IDE. In general, the requirements desired from an IDE for web service composition can be summarized along the following dimensions:

R.1 Functionality: The IDE has to encompass the entire process of web service composition starting from requirements gathering to creation of the composite service to deployment.

R.2 Support for multiple Users: Due to its scope extending from developmental to runtime aspects, the IDE needs to cater to different kinds of users including service developers, service deployers, and administrators.

R.3 Interface: The IDE has to present a unified view of different stages of software development. Specifically, any hard boundaries between the development, deployment and runtime stages of the composition process need to be evened out for enabling smooth transition from one stage to the next.

R.4 Integration: The IDE needs to deal with a set of disparate technologies such as OWL/OWL-S, WSDL and BPEL etc., while hiding all the heterogeneity from the user. This is important since different communities have developed solutions to different but related problems and all these need to come together in a unified manner. The integration involves providing support for different kinds of editors, viewers and registries. Furthermore, the information content in all of these should present a consistent view of the composition scenario being worked upon.

R.5 Usability: The IDE needs to provide most commonly used controls and menu options and enable all the complex functionality through these. This is to help create a familiar environment with most tasks organized in an intuitive fashion.

R.6 Adaptation: Since the IDE deals with runtime aspects of the composition as well, it needs to be adaptive to faults and other runtime changes.

R.7 Scalability: The IDE should work well with a small set of services (for example, inside an enterprise) as well as with a huge number of services (for example, in the Internet environment).

R.8 Security: Given that the IDE deals with development and even deployment of new business functionality, there is also a need for an appropriate access control mechanism. This would ensure that only authorized users are allowed to perform such critical operations.

We now describe Synthy IDE, which has been designed keeping the above mentioned requirements in mind. It is based upon the two-stage service composition approach described in the previous section. As we go along, we shall indicate the requirements being fulfilled by various design choices. Figure 2 shows an annotated version of the opening screen of Synthy IDE. It is implemented as a plug-in to the

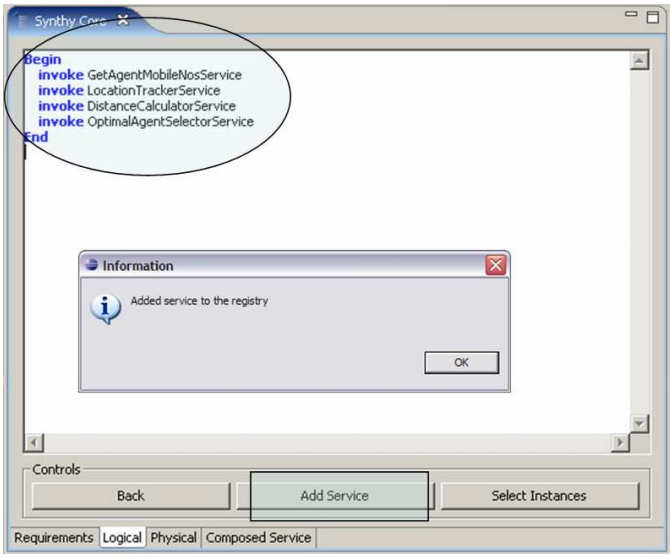

Figure 3: Logical Tab

Eclipse platform[1] (**R.4, R.5**) [2]. Eclipse is an open source *meta* IDE built using Java and can be used for building other IDEs. By becoming part of the Eclipse framework, the Synthy IDE becomes immediately usable by the large community of Eclipse users and the plugin based approach allows to tap into various existing and future components being added to the Eclipse platform. The central view labeled as *Synthy Core* is the main view that provides an interface to most of the composition tasks. It is surrounded by several helper views that let the developer access a variety of information that may be needed while composing.

To enable smooth navigation between the development, deployment and runtime stages, the IDE provides the user with a guided traversal of the composition process (**R.2, R.3**). For this purpose, it implements a wizard like feature with the help of tabbed panes. There are four tabbed panes in Synthy Core, labeled *Requirements, Logical, Physical and Composed Service*. Each of these enables a particular stage in the end-to-end web service composition process (**R.1**) and has a control button that guides the user to the next stage as a logical step from the current activity (**R.3, R.5**).

The Requirements tab lets the user give the specification of the new service in terms of its functional parameters (**R.1**). It has four panes to accept Inputs, Outputs, Preconditions and Effects (IOPE) that define the new service. The Synthy Core in Figure 2 shows IOPEs being specified for Location Based Agent Selector service in the Helpline scenario. The tab lets the user view the new service specification in OWL-S (**R.4**), generated from IOPEs specified by the user. Choosing the 'Create Service' option here transitions to the Logical tab. A conscious design choice that we made was to accept partial specification at this stage, in terms functional requirements alone and accept the non-functional requirements later in the process.

[1] http://www.eclipse.org

[2] R.x references in round brackets indicate requirements fulfilled by the design choice being considered.

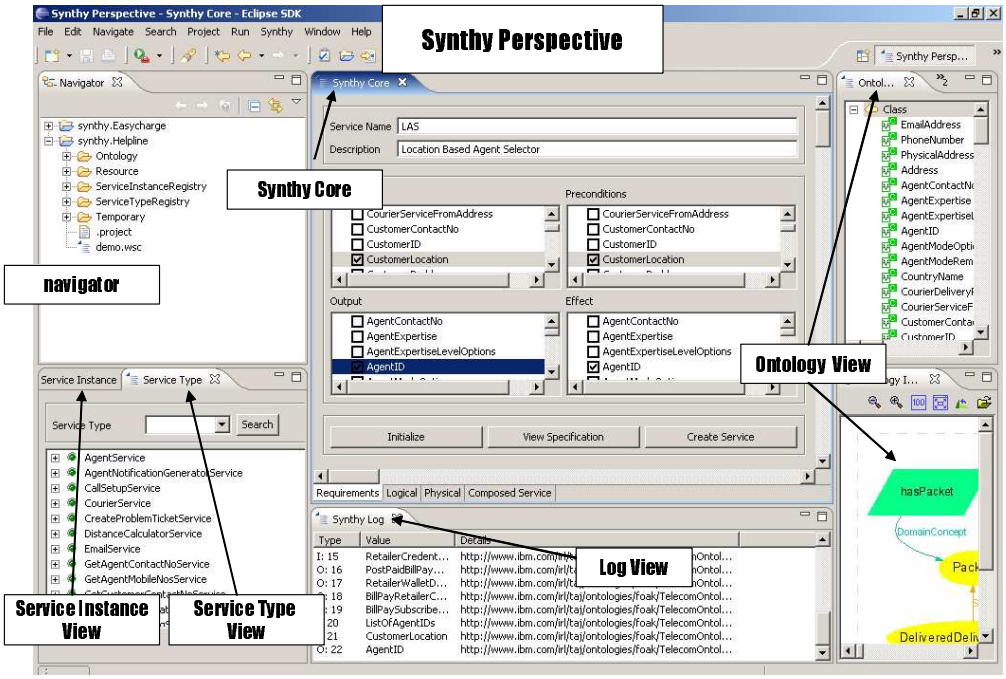

Figure 2: Synthy IDE for Web Service Composition

This choice helps the user focus on the immediate task at hand, i.e. functional composition, and requires enough input to get started (**R.3**). Further input is requested only if the initial stages have fruitful results (**R.7**). This makes the solution more scalable since the underlying composition problem to be solved is reduced to searching for solution in a smaller set of service types rather than potentially huge set of all service instances. Moreover, this also makes it possible to enable a role based usage of the IDE in situations where different people may be responsible for different portions of the composition process (**R.2**). An analyst might be concerned with composing the new functionality without worrying about other details. A service developer on the other hand would take that functional composition and use the IDE to create a deployable composite service that meets the specified QoS guarantees. This approach enables control points for validation by administrators and allows placeholders for inserting appropriate access control modules at the points where one role takes over from another (**R.8**).

The Logical tab deals with generation of the abstract workflow (**R.1**) using AI Planning techniques [16] (**R.4**). It displays the abstract workflow generated and lets the user add it to the registry as a new composite service type. Addition to registry establishes the newly created service type as an available component that can be reused later, either for direct invocation or for further composition. This way new and more complex services can be built from simpler, existing services. Figure 3 shows that the abstract workflow generated for Location Based Agent Selector service is being added to the service registry. From the Logical tab, the user has a choice to start all over again or to proceed to the Physical tab by clicking on the 'Select Instances' button (**R.3, R.5**).

The Physical tab handles specification of non-functional parameters for selection of web service *instances* for each component type in the newly created composite service (**R.1**). Figure 4 shows some non-functional parameters being specified for the Location Based Agent Selector service. These non-functional parameters are specified in terms of QoS guarantees expected from either the individual component services or from the composite service as a whole. Once done, the user can proceed to generate the BPEL which involves invocation of a QoS Solver module (**R.4**). The goal of this module is to select the service instances corresponding to each component service type so as to optimize the end–to–end QoS of the composed service subject to (QoS) constraints specified by the user [19]. Currently the IDE supports three QoS parameters, namely, availability, cost and response time. On a successful assignment, the IDE transitions to the 'Composed Service' tab (**R.3**), else it displays an appropriate message prompting the user to try again (with some constraints relaxed) (**R.5**).

The Composed Service tab lets the user verify and manipulate the composite service just created (**R.1**). This could include checking that the instances selected meet the specified non-functional requirements, as well as creation of data-flow [8]. Once satisfied, a user can proceed to deploy the composite service onto a runtime infrastructure (**R.2, R.3**). Figure 5 shows the WSDL generated for the composed Location Based Agent Selector service. Figure 6 demonstrates the reuse of this service as a component in the complete Helpline service. Following this bottom-up approach, a developer can first build smaller components and then reuse these to form higher level components. This would essentially result in a library of service types that get used not only to realize the complete solution (such as the helpline service) but also become reusable assets for future

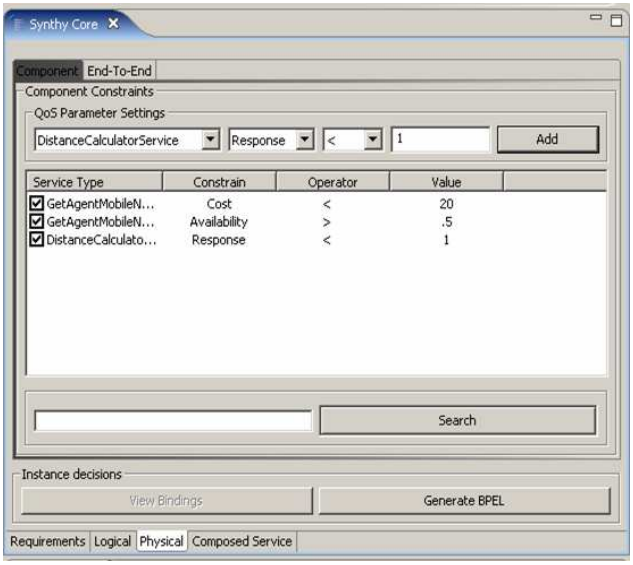

Figure 4: Physical Tab

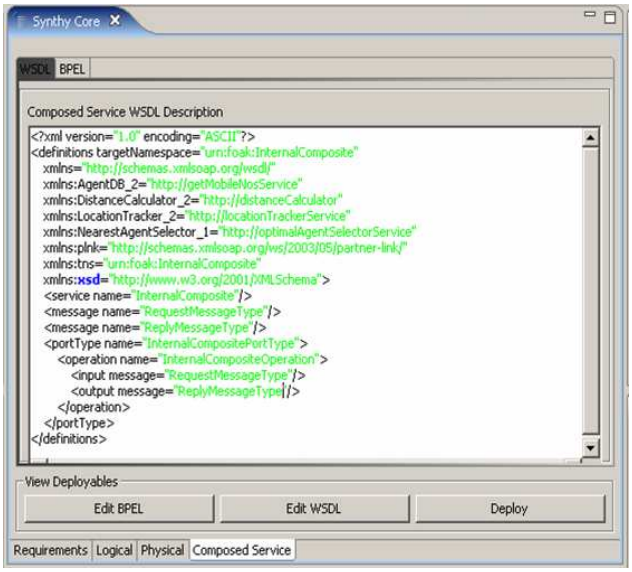

Figure 5: Composed Service Tab

needs.

The helper views complement the Synthy Core by capturing and maintaining the context of the composition process from start to finish (**R.1, R.3**). There are four types of helper views as shown in Figure 2. These include the *Navigator view*, *Registry views*, *Ontology views* and *Synthy Log view*. The Navigator view displays the resource tree representing the organization of various files on disk associated with the composition project. The Registry views connect to different repositories. It has two sub-views - Service Type view and Service Instance view, which show a list of available service types and instances respectively. Service type view provides ways for exploration of the available service types as well as for querying them. All IOPEs of each service are displayed within this view. The Service Instance view connects the two repositories by

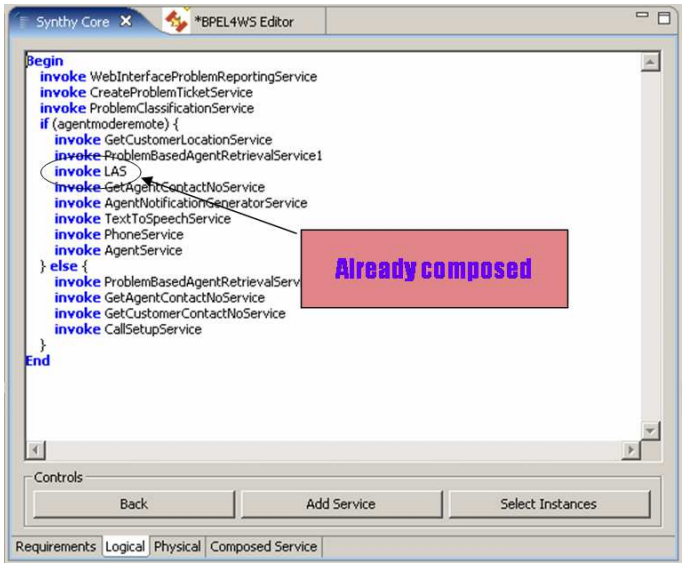

Figure 6: Reuse of the Composed Service

allowing service instances for a particular service type to be searched for. The Synthy Log view displays status information related to user interaction in the *Synthy Core*. Finally, two Ontology views on the right side enables the user to browse through the concept ontology as well as view it graphically.

From a *solution integration* point of view, the IDE is built upon several modules each of which implements a key technology enabling the end to end solution. The abstract workflow creation requires an AI Planner [14], whereas deployable workflow creation requires an IP Solver for selection of appropriate service instances. Similarly, the specification, storage, and retrieval of service specifications require an ontology management system[3] as well as a service instance registry [6] (**R.4**).

Out of the requirements listed in the beginning of this section there are few which are not yet fully addressed by Synthy IDE. Specifically, we identified some fault resilience techniques [4] that could enable the IDE to adapt to changes and faults in the runtime environment (**R.6**). These have not yet been integrated in the Synthy IDE. Similarly, the design of the tool has appropriate points for incorporation of access control but these features have not been enabled yet. Finally, scalability of the IDE can be enhanced by adding support for more than one service type registry and service instance registry.

4. ECLIPSE PLUG–IN DEVELOPMENT

The Synthy Integrated Development Environment (Synthy IDE) is implemented using Eclipse framework [7]. Its architecture supports extensibility by providing well defined extension points where other *plug–ins* can add functionality. A plug-in is the smallest *pluggable* component identified by the Eclipse framework. We can leverage the work done by other developers by integrating their plug–ins with our tool.

[3]http://www.alphaworks.ibm.com/tech/snobase

This feature is particularly useful for developing a complex IDE such as the one for service composition which requires support for multiple functionalities. Furthermore, Eclipse offers a unique way of loading plug–ins; loading of a plug–in is delayed until a component or functionality of that plug–in is requested. This allows the IDE to be decomposed into small plug–ins; each plug–in being loaded only when required.

The overall plug–in architecture of Synthy IDE with extension points is shown in Figure 7. It includes a generic set of

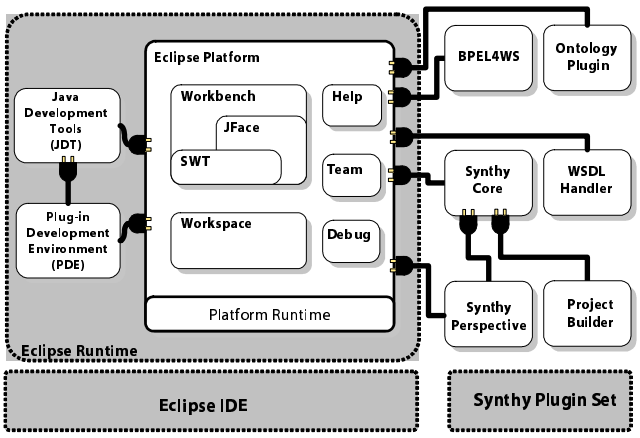

Figure 7: Plug–in architecture of Synthy IDE

components from the Eclipse IDE and a Synthy specific set of components indicated by *Synthy Plug–in Set*.

4.1 Architecture and Organization

The Synthy Plug–in set broadly consists of Synthy Core - the central part of the IDE where the new service is actually composed, a set of wizards for creating projects and gathering resource information, and Helper Views to provide developer with all the information necessary for composition. The Synthy Perspective binds all of these together into a single integrated framework serving the goal of providing a *unified interface* across the composition process.

Synthy Core: This is the main studio of the service developer (refer Figure 2). It initiates loading of the concept ontology, the service type registry as well as the instance registries. It also coordinates the invocation of the Planner during the Logical stage, the optimization routine during the Physical stage etc.

The Synthy Core utilizes the functionality of different editors at different steps of the composition process. These include *Synthy Core Editor*, *BPEL Editor*, *WSDL Editor*, *Binding Editor*, *Instance Editor* and the *Default Text Handler*. All of these extend the `MultiPageEditorPart` class available in the Eclipse PDE (Plug-in Development Environment) and multiple instances of these editors can be opened at the same time. This allows the tool user to get the complete picture at any moment by viewing and analyzing different resources involved in the composition process. This also allows multiple service compositions to be active simultaneously unlike a wizard that could allow focusing on a single task at hand.

The Synthy Core Editor presents a tab based environment capturing the stages of the composition process right from specification to deployment. Separate tabs are provided for the four stages, *Requirement* $\longrightarrow$ *Logical* $\longrightarrow$ *Physical* $\longrightarrow$ *Composed service* thus implementing the complete composition process. Tab browsing capability and context preservation during transition from one tab to the next allows user to return back to any stage, refine or review, and proceed further. For instance, given a set of QoS requirements, if no concrete composition can be generated, the developer can return to the previous stage and relax the constraints specified earlier. This improves the *usability* of the tool drastically since the developer need not start all over again from scratch and can follow an iterative approach for composition.

Wizards: Tasks such as new project creation that are repetitive and may involve a sequence of activities to be performed are made simpler in the IDE through the use of guided user dialogs or wizards. The project creation wizard called *Synthy Project Builder Wizard*, allows developer to specify locations of all the necessary resources such as the *concept ontology*, the *service type ontology*, the *instance registry*, and the *resource* file containing rules for ontology processing. The tool contains another wizard called *WSC Wizard* that captures information required for setting up a new service composition.

The functionality encoded in Synthy Core and the wizards helps hide the heterogeneity of the technologies working underneath and presents an *integrated view* with a *simple user interface*.

Helper Views: As discussed in Section 3, these enable the developer to easily access the information required for composing a new service. The Navigator view facilitates interaction with resources initially gathered through wizards and those added during the process of composition. It displays resource tree representing the organization of the project. The Registry and Ontology views enable visualization of service types, services instances and ontological terms that the tool user needs to refer to and use at different stages of composition. These views, therefore, further hide the heterogeneity and complexity of underlying technologies while presenting a *consistent view* of context information as the composition progresses in the Synthy Core. The helper views are developed using various existing plug–ins demonstrating reuse of and easy integration with available tools. The plug–ins used include the WSDL plug–in, BPWS4J plug–in and the OWL Ontology plug–in.

Synthy Perspective: Synthy Core and Helper Views are encompassed in a single shell using a facility provided by Eclipse known as a *perspective*. A perspective is a visual container for a set of views and editors (parts) and enables integration among them. The Synthy perspective controls the visibility of items in the model and the user interface and determines what can be seen in the model (projects, folders, or files) and what can be displayed in the user interface (views).

4.2 Component Design and Integration

Synthy IDE is realized through component plug–ins developed in-house as well as existing components all of which are integrated together into a single unifying framework.

We developed an *Ontology Viewer* plug–in to enable visualization of terms and properties present in the concept ontology. These are used in Logical phase for specifying the Input, Outputs, Precondition and Effects of a service type description. The ontology itself along with service type registry was maintained using SNOBASE[4] – an ontology management system. Service instances in our tool are maintained in a Web Service Matchmaking Engine (WSME) registry [6]. A WSME plug–in for eclipse allows administration of the instances registry by providing capabilities to browse, search and advertise services. The advertisement of a service instance includes specification of QoS parameters as well as a link to the corresponding WSDL. This plug–in was used to enable the Registry View for service instances.

In Physical phase, selected service instances corresponding to service types in a newly composed service are visualized through a *Binding Viewer* plug–in that we developed for this purpose. To enable display and editing of WSDL and BPEL outputs generated by our tool we leveraged existing plug–ins. Specifically, for visualizing and editing the BPEL4WS specification generated for the composite service, we used the BPWS4J plug–in [5]. It includes features such as context sensitive menus to facilitate editing business processes, synchronized XML source and tree views of the business processes being created etc.

The different component plug–ins mentioned above were integrated in various ways based upon the requirements of the tool. Situations where different resources (such as WSDL , BPEL, OWL files etc.) were required to be handled, appropriate editor plug–ins were *invoked* from the same module. Common data was shared among Synthy Core and other views through tool specific APIs. For instance, the concept ontology is loaded by Synthy core and shared with other plug–ins by encapsulating it in an object. This not only avoids performing the computation heavy ontology load operation again but also presents a consistent picture through all the plug–ins. Any changes are published by the Synthy Core to other views, in this case. Finally, the User Interface components such as wizards and Views support the internal integration to help present a unified IDE rather than a set of incompatible components stitched together.

5. SYNTHY IN USE

Synthy has been demonstrated at leading research forums as well as to a number of customers. It has been used to compose services in a number of pedagogical scenarios. The Helpline scenario (to automate a typical Helpline of a consumer goods manufacturer) is a medium-scale industry–relevant scenario with 25 web service types, 2-4 web service instances per service type, and about 100 terms in the ontology. The composition for this case takes less than a second for both logical and physical stages. We have another pedagogical Flower Shop service with 5 web service types and about 60 terms in the ontology.

We also investigated how Synthy may be used in industry during a collaborative project with one of the largest private telecom providers in the world. The customer explored use of Synthy to create new services in both the individual user–services domain (e.g., ringtones, video) or new infrastructure services (e.g., bill payment). A proof-of-concept was implemented and its ability to dynamically compose telco services was demonstrated.

Recently, we are actively involved in the process of making Synthy available from our organization's external web site so that service developers can download the IDE and try it out. We hope to include the site information in the paper at the time of publishing.

Finally, we are also undertaking an effort to identify and add features in the IDE that would make it more usable and reliable for service composition. We discuss some of the survey findings in the remainder of this section.

Preliminary Survey Results

We conducted a preliminary user survey of Synthy involving ten IT professionals. Our approach was to ask the participants about their familiarity with web services and semantic web, demonstrate web service composition using the tool, and then ask for feedback about the tool and its applicability to SOA. Eight out of ten people that we surveyed were familiar with web services and thought that web service composition is useful. Out of these eight, two were experts in web services, three moderately familiar, three novice. The rest two didn't know anything about web services. All those who were familiar with web services felt that semantics helps in service composition. However, none of the participants were aware of any such tool for web service composition.

Some of the key findings which emerged out of the survey are as follows. Seven out of ten people felt that the tool is user friendly. Eight out of ten people thought that the output of the tool is useful in terms of deployment on a run–time engine. However, no clear answer came out of the survey regarding this tool enabling faster time to market for new services. One person who earlier (before the demonstration) didn't think that semantics helps in service composition changed his opinion after the demonstration. Half the people thought that this tool enables SOA, while two were neutral and the rest did not know. Finally, the participants thought that developers and business process analysts would find this tool most useful. The analysts can use it to define a template of the composite service and the developers or deployers can then use the physical stage to ground the bindings for each of the component services.

In summary, the results were quite encouraging with respect to the group surveyed. However, we consider them as initial feedback to be followed by a full–fledged user survey. We also got some useful suggestions regarding the enhancements that can be made to the tool. Some of these are incorporation of hierarchical browsing and searching of the ontology to give the service specification, supporting expressions to specify IOPEs of the composite service at the logical

[4]http://www.alphworks.ibm.com/tech/snobase
[5]http://www.alphaworks.ibm.com/tech/bpws4j

composition stage, support for visual composition(drag and drop), automatic deployment of the output of the tool, and adaptation of the composition to the changes in the run–time environment. We plan to work on some of these in future.

6. RELATED WORK

In their survey of web services composition approaches, [5] note the importance of a service composition middleware to support composition in terms of abstractions and runtime infrastructure. The main elements of such a middleware, they identify, are a composition model and language to specify the services involved in the composition, a development environment with a graphic user interface to browse components, and a run-time environment to execute the business logic. However, the authors point to the scarcity of such middlewares for even manual composition approaches. An example of such a middleware is from HP in which E-flow [3] services are specified and then executed on an execution engine like HP e-speak. In an E-flow service, the process specification and dependency information is given along with information about exception handling, ACID transactions and security management. Another example is the SELF-SERV [12] platform in which service providers can register their Web services and code interface stubs, service composers can edit and deploy composite services, and end users can locate Web services and execute their operations.

A number of piecemeal solutions exist in the literature that address automatic (or semi–automatic) composition of Web services. The Mindswap group has developed a prototype [13] that presents to the user a set of matching web services at each step of the composition process. These matches are filtered using semantic description of the services, with the human controller making the final selection at each step. OWL-S IDE [6] provides a uniform integrated development environment that supports a semantic web service developer through the process of Java generation, to the compilation of OWL-S descriptions to final deployment and registration with UDDI. WSCE [18] provides a flexible environment for autonomic modeling and simulation of business processes. WSCE helps a BPEL programmer to test both application's functionality and non-functionality performance.

7. CONCLUSIONS

In this paper, we presented the design and development of an Integrated Development Environment (IDE) for end-to-end composition of Web services. The design of the IDE is based on a two staged service composition approach that separates the functional and non–functional requirements of the service being composed. While developing the IDE, we have attempted to synergize the strengths of disparate technologies, i.e. both Semantic Web technologies (like OWL, OWL-S) and distributed programming techniques (like WSDL, BPEL). Particular attention was paid to the fact that the IDE does not burden a user with the underlying complexities of the different technologies used, thereby making it simple to use by developers and business analysts alike. Finally, we have initiated the process of a methodological survey of the IDE by potential users, and will continue to incorporate their feedbacks in enhancing the proposed solution.

[6]http://www.daml.ri.cmu.edu/tools/details.html

8. REFERENCES

[1] V. Agarwal, G. Chafle, K. Dasgupta, N. Karnik, A. Kumar, S. Mittal, and B. Srivastava. Synthy: A System for End to End Composition of Web Services. *J. Web Semantics, Vol.3:4*, 2005.

[2] V. Agarwal, K. Dasgupta, N. Karnik, A. Kumar, A. Kundu, S. Mittal, and B. Srivastava. A Service Creation Environment based on End to End Composition of Web Services. In *Proceedings of the 14th International World Wide Conference*, May 2005.

[3] F. Casati and M. Shan. Dynamic and adaptive composition of e-services. *Information Systems*, 6(3), 2001.

[4] G. Chafle, K. Dasgupta, A. Kumar, S. Mittal, and B. Srivastava. Adaptation in web service composition and execution. In *4th International Conference on Web Services*2006.

[5] S. Dustdar and W. Schreiner. A Survey on Web Services Composition. In *International Journal of Web and Grid Services, Vol. 1, No.1 pp. 1 - 30*, 2005.

[6] C. Facciorusso, S. Field, R. Hauser, Y. Hoffner, R. Humbel, R. Pawlitzek, W. Rjaibi, and C. Siminitz. A Web Services Matchmaking Engine for Web Services. In *Proceedings of 4th Intl. Conf. on e-Commerce and Web Technologies*, September 2003.

[7] T. E. Foundation. Eclipse framework. `http://www.eclipse.org/`, 2006.

[8] A. Kumar, B. Srivastava, and S. Mittal. Information Modeling for End to End Composition of Web Services. In *International Semantic Web Conference*, 2005.

[9] A. Patil, S. Oundhakar, A. Sheth, and K. Verma. Meteor-s web service annotation framework. In *Proceeding of the World Wide Web Conference (WWW 2004)*, July 2004.

[10] S. Ponnekanti and A. Fox. SWORD: A Developer Toolkit for Web Service Composition. In *Proc. of the 11th International World Wide Web Conference*, 2002.

[11] S. Staab et al. Web services: Been there, done that? *IEEE Intelligent Systems*, pages 72–85, Jan-Feb 2003.

[12] Q. Sheng, B. Benatallah, M. Dumas, and E.-Y. Mak. Self-Serv: A Platform for Rapid Composition of Web Services in a Peer-to-peer Environment. In *Proceedings of the 28th VLDB Conference*, 2002.

[13] E. Sirin, J. Handler, and B. Parsia. Semi-automatic Composition of Web Services using Semantic Descriptions. In *Proceedings of Web Services: Modeling, Architecture and Infrastructure workshop, in conjunction with ICEIS2003.*

[14] B. Srivastava. A Software Framework for Building Planners. In *Proc. Knowledge Based Computer Systems (KBCS 2004)*, 2004.

[15] B. Srivastava and J. Koehler. Web Service Composition - Current Solutions and Open Problems. ICAPS 2003 Workshop on Planning for Web Services, 2003.

[16] D. S. Weld. Recent Advances in AI Planning. *AI Magazine*, 20(2):93–123, 1999.

[17] D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau. Automating daml-s web services composition using shop-2. In *Proceedings of International Semantic Web Conference*, 2003.

[18] X. Yu, L. Zhang, Y. Li, and Y. Chen. WSCE: A Flexible Web Service Composition Environment. In *Proceedings of International Conference on Web Services*, 2004.

[19] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Z. Sheng. Quality Driven Web Services Composition. In *Proceedings of 12th International World Wide Web Conference*, May 2003.