RI 04008 16 November 2004 Computer Science

IBM Research Report

A Service Creation Environment based on End to End Composition of Web Services

Vikas Agarwal, Koustuv Dasgupta, Neeran Karnik, Arun Kumar, Ashish Kundu, Sumit Mittal and Biplav Srivastava
IBM Research Division
IBM India Research Lab
Block I, I.I.T. Campus, Hauz Khas
New Delhi - 110016. India.

A Service Creation Environment based on End to End Composition of Web Services

Vikas Agarwal, Koustuv Dasgupta, Neeran Karnik, Arun Kumar
Ashish Kundu, Sumit Mittal and Biplav Srivastava
IBM India Research Laboratory
Block 1, IIT Campus, Hauz Khas
New Delhi 110016, India
Email: {avikas,kdasgupta,kneeran,kkarun,kashish,
sumittal,sbiplav}@in.ibm.com

Abstract

The demand for quickly delivering new applications is increasingly becoming a business imperative today. Application development is often done in an ad hoc manner, without standard frameworks or libraries, thus resulting in poor reuse of software assets. Web services have received much interest in industry due to their potential in facilitating seamless business-to-business or enterprise application integration. A web services composition tool can help automate the process, from creating business process functionality, to developing executable workflows, to deploying them on an execution environment. However, we find that the main approaches taken thus far to standardize and compose web services are piecemeal and insufficient. The business world has adopted a (distributed) programming approach in which web service instances are described using WSDL, composed into flows with a language like BPEL and invoked with the SOAP protocol. Academia has propounded the AI approach of formally representing web service capabilities in ontologies, and reasoning about their composition using goal-oriented inferencing techniques from planning. We present the first integrated work in composing web services end to end from specification to deployment by synergistically combining the strengths of the above approaches. We describe a prototype service creation environment along with a use-case scenario, and demonstrate how it can significantly speed up the time-to-market for new services.

1 Introduction

The demand for quickly delivering new applications is increasingly becoming a business imperative today. For example, given the intense competition in the telecom sector, mobile telephony service providers need to continually develop compelling applications to attract and retain end-users, with quick time-to-market. Often, if a competitor introduces a new service, the service provider must offer a similar or better service within days/weeks, to avoid losing customers. Also, a service provider can attract enterprise customers by offering custom-developed value-added services that leverage its telecom and IT infrastructure. Enterprise customers typically offer significantly higher margins than consumers, and are thus more attractive. Service providers therefore need tools and standards-based runtime platforms to quickly develop and deploy interesting applications for their clients. This would assist in their transition towards "on demand", responsive businesses.

Much of this service/application development is currently done in an ad hoc manner, without standard frameworks or libraries, thus resulting in poor reuse of software assets. When a new service is needed, the desired capability is informally specified. An application developer must then create this capability using component services available in-house or from known vendors. This process is essentially manual. For example if a mobile service provider wishes to offer a taxi-request service to its users, the developer must pick a third-party taxi service (with an advertised network interface) apart from in-house services like location-tracking, accounting, etc. and design a workflow that delivers the required functionality. The dynamic nature of the environment impacts the development process as well. For example, new taxi services may become available, offering better and/or cheaper services; physical changes in the network or environment may necessitate a redesign of the flow, etc.

Web services have received much interest in industry due to their potential in facilitating seamless business-to-business or enterprise application integration[21, 26]. Web services offer standardized interface description, discovery and messaging mechanisms. Also, the programming tools and runtime environments for web services have now matured. A component-oriented software development approach where each software is wrapped as a web service would offer substantial benefits in the mobile service provider's scenario. Mobile user applications often rely on several, relatively simple building blocks – user profile look-ups, address books, location-tracking services, accounting and billing services, etc. Many of these building blocks are already in place, but they are not easy to reuse and integrate into new applications because they are not built using standardized frameworks or component models. This leads to high development costs, and substantial time-to-market for new services. This could be alleviated by building applications using the service-oriented architecture (SOA) paradigm, using web services as the underlying abstraction.

We find that two different approaches have been taken to standardize and compose web services. The business world has adopted a distributed systems approach in which web service instances are described using WSDL, composed into flows with a language like BPEL[1], and invoked with the SOAP protocol. Academia has propounded the AI approach of formally representing web service capabilities in ontologies, and reasoning about their functional composition using goal-oriented inferencing techniques from planning[15]. These approaches

[1] http://www.ibm.com/developerworks/webservices/library/ ws-bpel/

by themselves are piecemeal, and insufficient. The former has focused on the execution aspects of composite web services, without much consideration for requirements capture and the development process. The latter approach has stressed on the feasibility of service composition based on semantic descriptions of service capabilities, but its output cannot be directly handed off to a runtime environment for deployment.

In this paper, we demonstrate how web services composition can be leveraged for business process integration, by synergistically combining the strengths of both the above approaches. The main contributions are:

- To the best of our knowledge, we present the first *end to end* web services composition methodology that, given a formally specified requirement for a new service, stitches together semantically-annotated web service components in a BPEL flow that delivers the required function.
- We propose a principled two-stage web services composition approach leveraging the differentiation between web service *types* and *instances*. This helps in handling different goals, different data, different rates of change of data at each stage, and different means to optimize them. It allows us to achieve scalability.
- We describe an end to end working prototype: (a) Ontology matching, composition at type level with service matchmaking (b) Composition at physical level with instance selection (c) Deployment onto a decentralized workflow orchestration infrastructure.

The rest of this paper is organized as follows. In the next section, we describe a business process integration scenario and motivate the role of web services composition. We then describe our approach (Sec. 3) followed by details on its main aspects – logical composition (Sec. 4) and physical composition (Sec. 5). Section 6 illustrates our solution for the use-case scenario. In Sec. 7, we provide a summary of related work. Finally we conclude with some directions for future work.

2 A Motivating Example

Service providers, like telcos, are increasingly targeting businesses as customers because of the higher margins and longer-term relationships. Suppose a telco wants to enable an enterprise customer to use its telecom and IT infrastructure by creating and deploying services that automate the customer's business processes. As an example the telco is attempting to automate a typical Helpline (or call center) for a washing machine manufacturer.

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, we need to find an agent in the field who can visit the customer and fix the washing machine. The service provider would like to create a set of web services that automate parts of this process to whatever extent possible, and keep aggregating these components to create higher-level composite services. Once such a software infrastructure is developed, the telco could offer it as a service to various enterprise customers (appliance manufacturers, software vendors, etc), with minor customization. Figure 1 summarizes the workflow in this Helpline scenario.

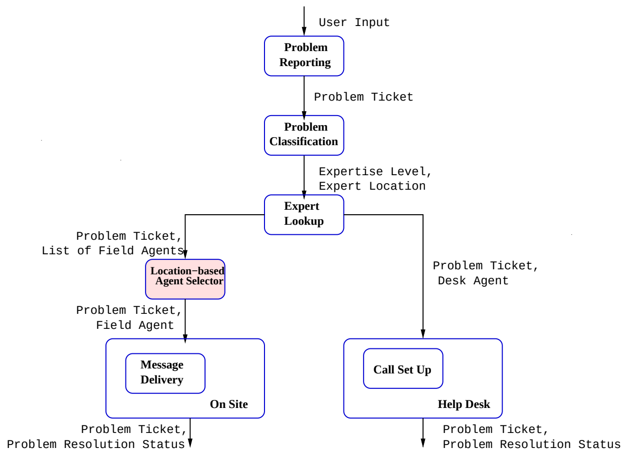

User Input
Problem Reporting
Problem Ticket
Problem Classification
Expertise Level, Expert Location
Expert Lookup
Problem Ticket, List of Field Agents
Location-based Agent Selector
Problem Ticket, Desk Agent
Problem Ticket, Field Agent
Message Delivery
On Site
Call Set Up
Help Desk
Problem Ticket, Problem Resolution Status
Problem Ticket, Problem Resolution Status

Figure 1: Helpline Service

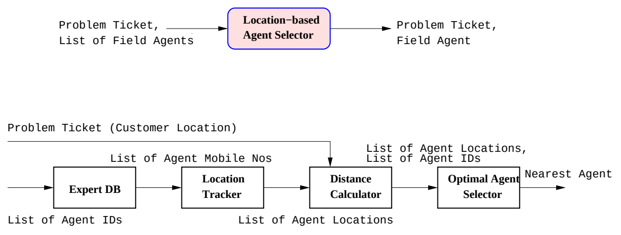

Problem Ticket, List of Field Agents
Location-based Agent Selector
Problem Ticket, Field Agent
Problem Ticket (Customer Location)
List of Agent Mobile Nos
List of Agent Locations, List of Agent IDs
Expert DB
Location Tracker
Distance Calculator
Optimal Agent Selector
Nearest Agent
List of Agent IDs
List of Agent Locations

Figure 2: Location-based Agent Selector Service

Here is a sampling of the component services that may be available in the service provider's infrastructure: Location tracking, SMS, Call Setup, Agent Expertise data, Problem Classification, Agent Selection. Some of these provide telco-specific functions such as delivering SMS text messages, location tracking of mobile phones, etc. Others are specific to the application domain, e.g. problem classification.

A developer needs to create a set of higher-level services using these components. Consider for example a location-based agent selector service (Fig. 2). Given a customer's location and a list of agents out in the field, this service needs to select one of the agents, based on proximity to the customer's residence. This selected agent will then be asked to visit the customer and fix her washing machine. The bottom half of Fig. 2 shows how this can be done by creating a flow linking together several component services, feeding them the right inputs, etc. Doing this manually takes time (and the developer has to know which components exist, and how to connect them up). Instead, we provide a tool that discovers the relevant services from amongst the available ones, and creates the control flow between them. The available services are *semantically* annotated, providing meta-information about their functionality in the context of a domain model. The developer only needs to (formally) specify the requirements of the service to be created. The tool can then generate a flow, and with some developer inputs, deploy the flow on to a runtime infrastructure. This should lead to quicker service creation, and thus faster time-to-market for new services.

Further, the newly created Location-based Agent Selector (LAS) service itself becomes available as a component. It can now be reused in creating other flows, such as the one in Fig. 1. Each new service thus enriches the infrastructure and makes the developer's task easier in future. We will use the LAS service as a running example to explain the phases in the composition process. Our service creation environment however includes a domain model and ontology for the entire Helpline Automation scenario, and we demonstrate the automated composition of the complete flow of Fig. 1 in Sec. 6.

3 System Overview

Our service creation methodology, based on web service composition techniques, consists of the following steps:

1. *Service Representation:* Representing the available services and their capabilities.

2. *Requirements Specification:* Specifying the desired functionality of a new service.

3. *Composition:* Constructing a composition of available services that provides the desired functionality.

4. *Composite Service Representation:* Representing the new composite service and its capabilities so that it can be programmatically deployed, discovered and invoked.

In previous work a similar process has been applied at different levels of abstraction, none of which individually yields a practical solution. Our system takes an end to end view that synergistically combines the AI approach and the distributed programming approach currently adopted by academia and the industry respectively. It drives the composition process right from specification of the business process, through creation of desired functionality using planning techniques, through generation of a deployable workflow by selection and

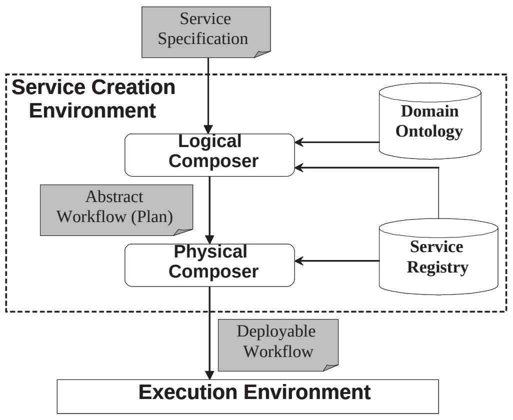


Figure 3: System Overview

binding of appropriate service instances, to finally deploying and running the composite service. This integrated solution achieves the best of both worlds and provides scalability to the composition process. We have built a service creation environment that realizes this approach in terms of the following phases of composition:

1. **Logical Composition:** This phase provides functional composition of *service types* to create new functionality that is currently not available.

2. **Physical Composition:** This phase enables the selection of component *service instances* based on non-functional (e.g. quality of service) requirements, that would then be bound together for deploying the newly created composite service.

This basic approach to automating the process of service creation is illustrated in Fig. 3. A **Service Registry** contains information about services available in-house as well as with participating 3rd-party providers. The *capabilities* of each available service *type* are described formally, using domain-specific terminology that is defined in a **Domain Ontology**. When a new service needs to be created, the developer provides a **Service Specification** to the **Logical Composer** module. Driven by the specified requirements, the Logical Composer uses generative planning-based automated reasoning techniques to create a composition of the available service types. Its goal is to explore *qualitatively* different choices and produce an abstract workflow, i.e. a plan (assuming a feasible plan exists) that meets the specified requirements.

In order to turn the plan into a concrete workflow that can be deployed and executed, specific instances must be chosen for the component services in the plan. The **Physical Composer** uses scheduling and compilation techniques in selecting the best web service instances to produce an executable workflow. The focus is now on *quantitatively* exploring the available web service instances for workflow execution. It queries the registry for deployed web service instances, to accomplish this task.

The workflow generated by the service creation environment must then be deployed onto a runtime infrastructure, and executed in an efficient and scalable manner. This is especially important in environments like that of a mobile service provider, where the number of end-users is likely to be very high. The state of the art is to execute the workflow using a workflow engine such as WebSphere Process Choreographer[2], with data flowing back and forth from this engine to the component web services. Our **Execution Environment** instead orchestrates the workflow in a *decentralized* fashion, with partitions of the flow executing concurrently, in network-proximity with the component services they invoke. These flow partitions are generated automatically by a Decentralizer tool, using static analysis of the input BPEL flow. The communication amongst these partitions is designed to minimize network usage, while retaining the original flow semantics. This, in conjunction with the added concurrency, results in better scalability and performance. For more details on our Execution Environment please refer to [5, 17]. In this paper, we will focus on the Logical and Physical composition stages.

4 Logical Composition

Figure 4 depicts our system for implementing the four steps of composition during Logical Composition. Available service types and their capabilities are represented in a Service Capabilities Registry. An Ontology captures the domain model. We use IBM's SNOBASE[3] as the management system for our ontology and the service capabilities registry. Specification of the desired service is supplied to a **Logical Composer** module that first gets it verified for syntactic correctness using a **Validator** module. The **Matchmaker** module allows querying the service registry for available services. Based upon the validated specification, **Planner4J** retrieves the set of candidate services using the matchmaker. The **Filter** module helps in pruning the set of candidate services before Planner4J uses planning techniques to create the composite service. We next discuss the issues that arise in each step of logical composition.

4.1 Representation of Service Types

To enable automatic discovery and composition of desired functionality, we need a language to describe the available web services. This can take place at two levels – web service types and web service instances. At the logical composition stage, the composition process typically involves reasoning procedures. To enable those, services need to be described in a high-level and abstract manner [11]. Therefore at this stage it suffices to describe the

[2]http://www.software.ibm.com/wsdd/zones/was/wpc.html
[3]http://www.alphaworks.ibm.com/tech/snobase

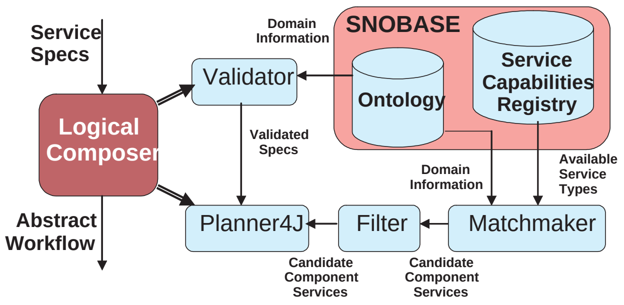


Figure 4: Logical Composition

capabilities of the *types* of web services, using semantic annotations. The second level of description becomes important in the physical composition stage where individual running services need to be identified for deploying the workflow. Once the language is known, the basic terms used in the language have to be drawn from a formal domain model. This is required to allow machine based interpretation while at the same time preventing ambiguities and interoperability problems.

The DARPA Agent Markup Language (DAML, now called OWL)[4] is the result of an ongoing effort to define a language that allows creation of domain models or *concept ontologies*. We use it to create the domain model using which services are described. The OWL-S markup language [18] (previously known as DAML-S) is also being defined as a part of the same effort, for facilitating the creation of *web service ontologies*. It specifies an *upper ontology* of services that defines the structure of a service description. It defines that a service *presents* a ServiceProfile (i.e. what the service does), is *describedBy* a ServiceModel (i.e. how it works) and *supports* a ServiceGrounding (i.e. how to access it).

Currently, OWL-S is designed to describe a single web service instance [18]. This is easily observed since the ServiceModel and ServiceGrounding aspects are specific to an instance of a web service. However, we believe that the *type* of a web service needs to be described independent of individual web service instances. This helps in working with large collections of web services – categorizing them, supporting multiple views, etc. [11].

We propose to separate the representation of web service type definitions from instance definitions. This means that the OWL-S upper ontology needs enhancements to have a *ServiceType* class hierarchy in addition to the *Service* hierarchy (see Fig. 5). The *ServiceProfile* model of the current OWL-S *Service* hierarchy is essentially a type definition and can be moved to the *ServiceType* hierarchy. The *ServiceProfile* of a *Service* could then point to the *ServiceProfileType* of *ServiceType* for structure, and contain the actual values of the Inputs, Outputs, Preconditions and Effects (IOPE) parameters applicable for that service instance.

[4]http://www.daml.org

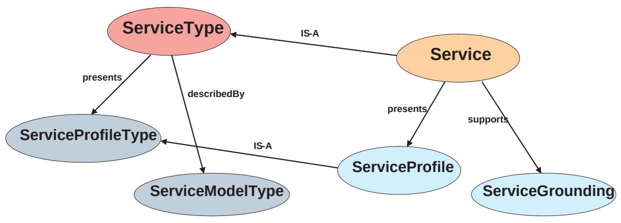


Figure 5: Modified OWL-S *upper ontology*

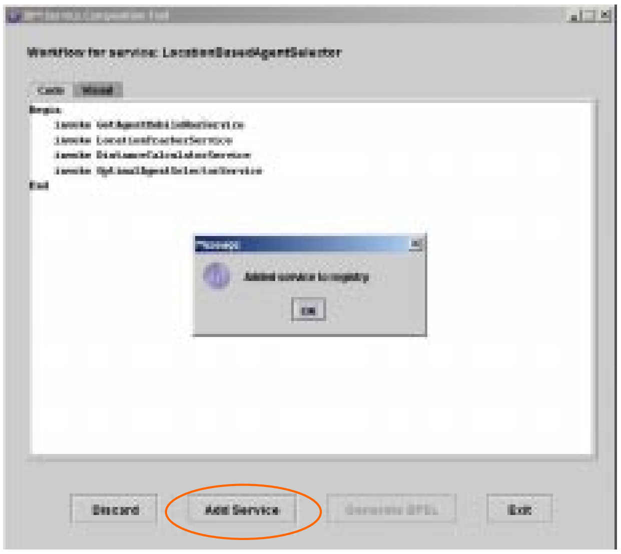
ServiceGrounding is a concept that applies to *Service* instances rather than types and can stay as it is. *ServiceModel* model is useful only if it describes the conversational aspect of the web service. In such as case, it should be included in the *ServiceType* hierarchy since the conversation model would be applicable to all instances of a service type. In other words, we propose to have an ontology for *ServiceType* that consists of *ServiceProfileType* and *ServiceModelType* model. This would be in addition to an ontology for *Service* instances that consists of a *ServiceProfile* and a *ServiceGrounding*. This approach has various modeling benefits. A new service type can be easily added into the ontology by creating an object of type *ServiceType*. This would include defining the parameters in its profile by populating the *ServiceProfileType* model, and describing the conversation model by populating the *ServiceModelType* model. Each service instance would be created as an object of type *Service* and include a reference to its *ServiceType* object. Its *ServiceProfile* model would contain the actual values of the parameters listed in the *ServiceProfileType* of that *ServiceType*. This approach of separating type definitions from instance definitions has been used successfully in data models for distributed systems management [16, 1].

However, in the absence of the model described above, we use the *Service-Profile* model of OWL-S to represent web service type definitions. Currently we do not deal with conversational systems and therefore do not require the OWL-S process model. The domain model from which all the concepts of the service ontology are obtained, is expressed in OWL. The task of providing descriptions for specific web service instances is deferred to the physical composition phase.

4.2 Requirements Specification

In order to create a new service, the developer should describe the required functionality as well as non-functional constraints such as throughput, response time, cost etc. We again use OWL-S for representing the requirements in IOPE terms, because the result of the composition is also a service. The developer is presented with a graphical user interface using which she can specify these requirements. The tool maps these to OWL-S for internal processing.

In keeping with our philosophy of qualitatively composing the plan before focusing on the quantitative optimization issues, the tool processes the requirements incrementally. The preconditions and effects are logical terms



Workflow for service: LocationBasedAgentSelector

```
Begin
    invoke GetAgentMobileNoService
    invoke LocationTrackerService
    invoke DistanceCalculatorService
    invoke OptimalAgentSelectorService
End
```

Added service to registry

Figure 6: Logical Plan for the LAS service.

and sentences, and are used during planning in logical composition. The inputs and outputs are expressions involving general data types (e.g. integers, strings, algebraic expressions) which are used during instance selection and flow concretization in the physical composition phase. It is possible to incorporate numeric inputs and outputs during planning as well – this approach to planning is called metric planning [6]. Exploring the feasibility of metric planning for end to end web service composition would be an interesting area for future work.

4.3 Composition through Planning

AI Planning deals with finding a course of actions that can take an agent from the initial state to a goal state, given a set of actions (legal state transformation functions) in the domain. Formally, a planning problem [29] P is a 3-tuple $\prec I, G, A \succ$ where I is the complete description of the initial state, G is the partial description of the goal state, and A is the set of executable (primitive) actions. An action sequence S (a plan) is a solution to P if S can be executed from I and the resulting state of the world contains G. A planner finds plans by evaluating actions and searching in the space of possible world states or the space of partial plans. Logical composition of web services can be cast as a planning problem by using the description of web services as actions, and forming initial and goal states from the specification of the service to be built along with the domain model [15].

For our service creation tool, planning for web services has some unique characteristics (refer to Fig. 4):

- The nature of planning is *limited contingency* planning. The value of all logical terms may not be known in the initial state (e.g. whether Agent needs to be desk-based or in the field) but they can be found at runtime using *sensing* actions. Plans of contingent planning problems have branches corresponding to different outcomes that sensing actions may find. However, the user may not be interested in all branches – which are exponential in the number of unknown terms – but only in specific branches. For the rest (usually unimportant or unlikely cases), the user may manually insert a default branch. We have implemented such a contingent planner in the Planner4J framework[24].

- *Filtering* is needed to remove irrelevant web services. Since the number of web service instances in a registry could be very large, the number of web service types can also be large. Given a goal specification, the **Filter** finds services of potential relevance to the goal without actually searching for the solution. Relevant services are those that can either contribute to the goals (atleast one effect unifies with a goal) or to the preconditions of any service which can potentially contribute to the goal.

- Our **Matchmaker**[7] matches the preconditions of a web service with the effects of another upfront. Another approach may be to perform matchmaking as needed during planning [22]. This can support more expressive matching (e.g. involving expressions of initially unknown terms) but at the cost of slower performance due to frequent reference to the ontology unifier.

- We provide *incomplete* plans on request. If no complete sequence of actions is possible from the available services for a given requirement, planning can still help the user scope down the composition request or point to missing capabilities in the ontology. The planner sorts the search space of non-solutions based on a heuristic distance to goal. The plan with the lowest such distance gives us a candidate plan for further development. This is especially valuable when the ontology development has not stabilized.

With these features planning is scalable, efficient and user friendly. We illustrate with some empirical results. On a contingent problem having a 7-step plan, with the filter enabled, the planner can return a solution in 4 seconds when 100,000 irrelevant service types/actions are present, whereas it takes an hour without the filter. If the user chooses specific branches, the planner can leverage it for better performance – in an experiment where 3 branches were specified on a contingent problem with 100 sensing actions (2^{100} possible branches), the planner takes 90% less time by leveraging this information rather than exploring the whole search space.

In Fig. 6, the planner was invoked for the LAS service. The initial state asserts that `CustomerLocation` is known and the goal is to find the agent (`AgentID`) nearest to the customer location. The output of the tool is a 4-step plan that can accomplish the goal. The created service is added to the service capabilities registry by the user.

4.4 The Abstract Workflow

The generated plan is a sequence of time steps, where each time step may have concurrent actions. Since plans can have branches which are contingent on specific conditions (called branch context) being met, actions are labeled with their context. The default context for an unconditional action is true, always valid.

The plan is translated to the workflow representation of BPEL, a language for expressing interactions and message exchanges between partner entities. It can be automatically interpreted and executed by a workflow engine. A BPEL specification can be abstract or executable depending on whether binding information has been excluded or included.

We render the generated plan as an *abstract* BPEL workflow since web service instance information is not known at this stage of the composition process. The actions in the plan are mapped to corresponding *invoke* activities in BPEL and organized into branches by inserting appropriate *switch* and *case* activities.

5 Physical Composition

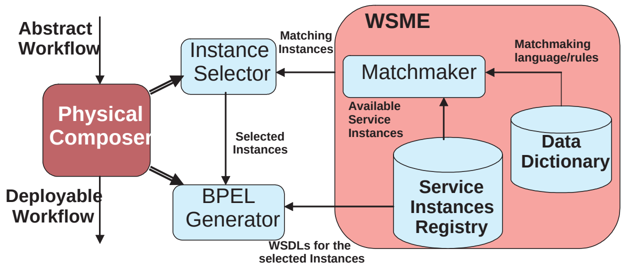


Figure 7: Physical Composition

In this phase, the abstract workflow for the composite service is fed to the Physical Composer, which *binds* each service in the workflow to a concrete service instance. The process of matching each service type to a corresponding instance, and then orchestrating between the set of instances in the resulting workflow is a non-trivial matchmaking problem. The problem has been addressed extensively for web services and involves a number of issues related to data flow orchestration, data type and invocation protocol matching, QoS matching and SLA composition. While some of these issues can be resolved in an automated manner, others might require manual intervention from a developer supervising the composition process. We next describe each of the steps involved in the Physical Composition stage, illustrated in Fig. 7.

5.1 Representation of Service Instances and Requirements

As in Logical Composition, we require a representation for service instances and composition requirements to facilitate Composition. It has been established that directory services, such as UDDI, are important but insufficient for this purpose[9] and need to be complemented with matchmaking facilities like symmetry of information exchange between services and their consumers, the ability of each party to describe requirements from the other party, a rich language to describe services' and consumer demands, and a methodology to choose efficiently among competing service instances.

To this end, we use the Web Services Matchmaking Engine (WSME) [9] – an engine capable of matching complex entities, and a **Data Dictionary** tool for defining the language for the matching process. Matching is performed between service instances and requirements specified by the consumer. In our case, the requirements come from the abstract workflow and additional matching criteria specified by the developer performing the service composition.

The engine is deployed as a Web service that receives queries and advertisements from the two parties involved in matchmaking. Each party essentially submits a description of itself and the demands from the other party. The Advertisement is submitted by the provider to WSME and is long lived, remaining in WSME until it is explicitly withdrawn by the provider or until the application server is stopped. The advertisement contains the following information: (1) MyType - this specifies the advertisement record-type. (2) YourType - this specifies the record-type expected to be submitted by the consumer query. (3) Properties - a list of the properties defined as MyType. Some of those properties may be defined as dynamic properties by the provider evaluated at runtime. (4) Rules (optional) - what the provider requires from the consumer.

A Query submission is sent from the consumer to WSME and is transient, terminating after initiating the matchmaking process and bringing it to its conclusion. The query contains the following information: (1) MyType - this specifies the query record-type. (2) YourType - this specifies the provider's advertisement record-type that the query is looking for. (3) Properties - a list of the properties defined as MyType. (4) Rules - what the consumer requires from the provider. The descriptions and demands can be dynamically created, deleted and modified in the form of properties and rules respectively, using a Data Dictionary tool.

The WSME rules allow both sides to select the other party they wish to deal with by specifying their eligibility. A rule is a WSME script that is evaluated at matchmaking time, resulting in a Boolean value. A rule can refer to the properties of the two parties whose advertisement and query are involved in the matchmaking process. Example of a possible consumer rule is the following: $return(my.MaxCost \leq your.cost)$. A problem arises if a rule refers to a property that was not supplied. To avoid such a situation, the WSME Type system defines the mandatory list of properties that a submission must provide;

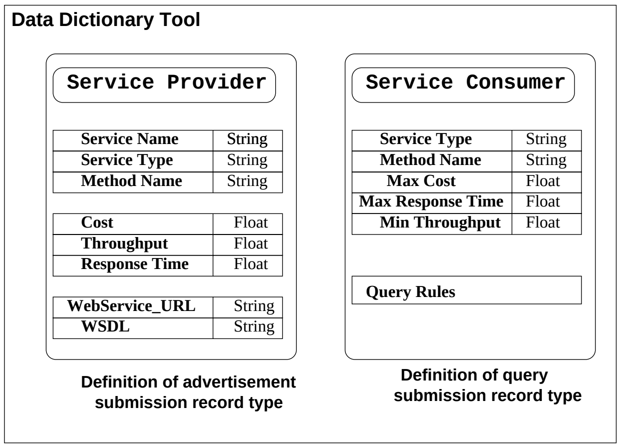


Figure 8: Advertisement and Query Formats for Helpline Service

the data dictionary contains those definitions.

We illustrate the Data Dictionary definitions used for composing the Helpline Service discussed in Fig. 8. Each service instance needs to advertise itself to the WSME service instance registry using the advertisement definition. Each advertisement record contains basic information like the service name, service type, method name and WSDL information, along with QoS-specific metrics like (expected) response time, throughput and cost of invoking the particular instance. Each query record specifies the method name and service type that needs to be bound to an instance along with additional rules that are specified by the developer supervising the composition process.

5.2 Matchmaking and Instance Selection

We employ a two-step approach where, in the first step, we use the WSME **Matchmaker** to select one or more candidate instances that match the requirement specifications. Since a specific service type can be matched with more than one instance, we next adopt a heuristic-based approach to select *one* among the conflicting instances, to complete the matchmaking process.

The WSME matchmaking process is a two-way or symmetric process - it brings together matching advertisements and queries by applying the rules of each party to the description of the other, thus allowing both parties to 'select' each other. A matching advertisement is called an *offer*. If more than one offer is available, they are collected together. Zero, one or more matching offers are sent to the consumer. For further details on the matchmaking process, the interested reader is referred to [9].

Next, we deal with those service types that have more than one matching offers (instances) from WSME. While there are many ways to perform the instance selection, we chose to employ a greedy heuristic approach to solve

```
<invoke name= "invoke−GetAgentMobileNumbers"
        partnerLink="link2"
        portType="GetAgentMobileNumbers"
        operation="GetAgentMobileNumbers"
        inputVariable="variable1"
        outputVariable="variable2"  / >

 <invoke name= "invoke−LocationTracker"
        partnerLink="link3"
        portType="LocationTracker"
        operation="LocationTracker"
        inputVariable="variable2"
        outputVariable="variable3"  / >

  <invoke name= "invoke−DistanceCalculator"
        partnerLink="link4"
        portType="DistanceCalculator"
        operation="DistanceCalculator"
        inputVariable="variable3"
        outputVariable="variable4"  / >

   <invoke name= "invoke−OptimalAgentSelector"
         partnerLink="link5"
         portType="OptimalAgentSelector"
         operation="OptimalAgentSelector"
         inputVariable="variable4"
         outputVariable="variable5"  / >
```

Figure 9: BPEL code for the LAS service

the problem. In particular, the **Instance Selector** finds instance binding assignments that optimize certain quality of service metrics. For the current prototype, we focus on commonly used QoS metrics like cost, response time and throughput. We assume that values of these metrics (as advertised in WSME) are statistically guaranteed. If a service type has multiple matching instances, we choose an instance based on the optimization criteria specified by the developer supervising the composition process.

5.3 BPEL Generation for Composite Service

Now that each service type in the abstract flow is bound to an instance, the **BPEL generator** produces a (concrete) BPEL workflow that can be deployed onto a runtime infrastructure, to realize the composite service.

We first generate the WSDL description for the composite service. It provides the name and interface of the composite service and describes the port types for stitching together the component services. Once the WSDL has been generated, partner link types are defined, linking the component services. The next step is the generation of the BPEL flow. Components are invoked in the manner described by the abstract workflow. The composite service accepts inputs from the user that is fed to the first component service and sends an output from the last component service back to the user. We introduce variables that capture the output of one service and provide it as input to the next. Specific details for each component service are obtained using the

WSDL description for the corresponding instance, present in the WSME service instance registry.

Though BPEL and WSDL are XML-based standards, we do not manipulate XML directly. We use an Eclipse Modeling Framework (EMF) model of BPEL (WSDL) that is automatically created from a BPEL (WSDL) schema[5]. The model provides in-memory representation of constructs and support for persistence to files (serialization) and loading from files (de-serialization). BPEL and WSDL manipulation become significantly simplified with the corresponding EMF models.

Note that the BPEL generated might not be readily deployable on a workflow engine. This is due to the fact that the code for messaging between component services needs to handle issues like (input/output) type matching and transformation, mismatch in invocation protocols that are used (synchronous vs asynchronous), ordering of parameters etc. While the BPEL workflow acts as the template for the composite service, it needs to be examined and possibly modified by the developer to ensure that the data flow between component services is handled properly. In the current prototype, this is done by allowing the developer to edit the BPEL workflow before it is actually deployed. We also make the observation that the handling of some of these matching problems could be delegated to the matchmaking engine (WSME), and we plan to investigate this approach in the future.

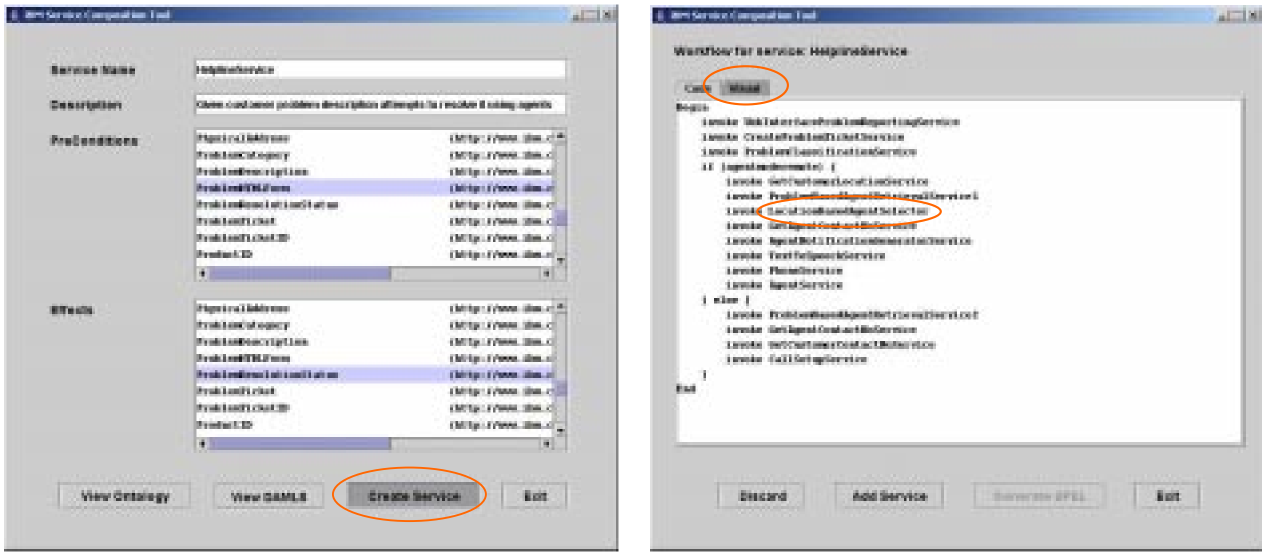
Figure 9 illustrates part of the BPEL code generated by the Physical Composer for the LAS service. It is composed of the four component services described in Sec. 2. Further, once physical composition is done, the WSDL description of this new service is added to the WSME instance registry, and can be later used in the composition of some other service.

6 Composing the Helpline Service

We now discuss how our service creation tool can be used for composing the Helpline service described in Sec. 2. Recall that the Helpline service consists of multiple components services like the LAS service, Message Delivery and Call Setup services. By way of running example, we showed how an executable workflow is created for the LAS service using the tool. The user can add the composite service to the service registry so that it is available for reuse.

For developing the Helpline service, the user may choose to use the tool to explore basic services available, build appropriate composite services, and finally build the Helpline service. Alternatively, the user could ask the tool to build the Helpline service at the outset using the available services, and let the tool search through the set of possible plans. We expect the user to prefer the former approach, when the scenario is large and the user wants to control the composition.

[5]http://www.eclipse.org/emf



IBM Service Composition Tool

Service Name: HelplineService

Description: Given customer problem description attempts to resolve it using agents

PreConditions:

```
PhysicalAddress              (http://www.ibm.c
ProblemCategory              (http://www.ibm.c
ProblemDescription           (http://www.ibm.c
ProblemHTMLForm              (http://www.ibm.c
ProblemResolutionStatus      (http://www.ibm.c
ProblemTicket                (http://www.ibm.c
ProblemTicketID              (http://www.ibm.c
ProductID                    (http://www.ibm.c
```

Effects:

```
PhysicalAddress              (http://www.ibm.c
ProblemCategory              (http://www.ibm.c
ProblemDescription           (http://www.ibm.c
ProblemHTMLForm              (http://www.ibm.c
ProblemResolutionStatus      (http://www.ibm.c
ProblemTicket                (http://www.ibm.c
ProblemTicketID              (http://www.ibm.c
ProductID                    (http://www.ibm.c
```

(a)

IBM Service Composition Tool

Workflow for service: HelplineService

```
Begin
   invoke WebInterfaceProblemReportingService
   invoke CreateProblemTicketService
   invoke ProblemClassificationService
   if (agentIsRemote) {
      invoke GetCustomerLocationService
      invoke ProblemBasedAgentRetrievalService1
      invoke LocationBasedAgentSelector
      invoke GetAgentContactNoService
      invoke AgentNotificationGenerationService
      invoke TextToSpeechService
      invoke PhoneService
      invoke AgentService
   } else {
      invoke ProblemBasedAgentRetrievalService2
      invoke GetAgentContactNoService
      invoke GetCustomerContactNoService
      invoke CallSetupService
   }
End
```

(b)

Figure 10: (a) Specifying input in the Composition Tool. (b) Logical Plan for Helpline Service.

We have approximately 100 terms in the ontology and 25 service types. Assume that the previously created composite LAS service has been added to the registry. Now the tool is invoked for the overall Helpline service with a precondition of `ProblemHTMLForm`, and the effect of `ProblemResolutionStatus` as shown in Fig. 10(a). The Logical Composer produces the plan shown in Fig. 10(b). Note that the LAS service is reused. The plan containing LAS service is selected over alternative plans without it, because the plan's heuristic cost is less[6]. Finally, the Physical Composer takes the abstract workflow and generates the appropriate BPEL (similar to Fig. 9).

7 Discussion and Related Work

The literature on web service composition is extensive, consisting of promising results and many challenges [27, 26]. To put it in perspective of this paper, we organize this section around design approaches for end to end composition, logical composition and physical composition.

[6]Designing heuristic functions so that user intent is respected is an active area of research in Hierarchical Task Network planning[8].

7.1 End to End Service Composition

In AI planning, the potential advantage of resource abstraction whereby causal reasoning is decoupled from resource reasoning is well-established[25]. Our work can be seen as applying the same idea to web services composition. Specifically, we differentiate web services at the twin levels of web service types and instances. Our phased approach is easier for the user to work with and limits the impact of frequent deployment and runtime changes on the goal-driven composition.

A planning-based phased approach has been used in [2] where an end-to-end system is described to construct workflows for manipulation of scientific data, which are executed on Grids. The domain involves composition at three levels – application domain level where appropriate applications are first selected, then an abstract plan is built with a planner, and finally it is detailed based on grid execution details. Two main differences with our work are that (a) they do not use ontologies while they recognize the need, and (b) the plan/workflow representation is simpler during logical composition – sequential, while we can handle branches as well. As our output is in BPEL which also has support for loops, exceptions and other behavioral constructs, the physical stage can be even more expressive.

In [28], executable BPELs are automatically composed from goal specification by casting the planning problem as a model checking problem on the message specification of partners. The approach is promising but presently restricted to logical goals and small number of partner services. In contrast to our top-down approach, Mandell and McIlraith[12] extend a BPEL engine to support runtime service selection using a semantic discovery server.

7.2 Logical Composition

The literature on composing services based on annotations (semantically organized in ontologies or otherwise) has taken two paths. One direction is disambiguating similar annotations using domain meta-data, rules, etc. The other direction is on methods to combine services whose annotations match based on some notion of similarity.

In [19], matching of web services from a directory is formalized based on various inexactness measures. In [11], the authors have identified the information that a Semantic Web Service must expose in order to fulfill the objective of automated discovery, composition, invocation and interoperation.

SWORD [20] was one of the initial attempts to use planning to compose web services. It does not model service capabilities in an ontology but uses rule chaining to composes web services.

Sirin et al. [23] use contextual information to find matching services at each step of service composition. They further filter the set of matching services by using ontological reasoning on the semantic description of the services as well as by using user input. They attempt to overcome lack of support for service types in OWL-S by creating a class hierarchy of Service Profiles. A new sub-class is created for each value of an IOPE parameter.

There are three problems with their approach. First, a large set of values for an attribute of a service would result in generation of that many classes. Second, to represent a functionality with multiple attributes a huge number of services, one each for a set of possible values of all attributes, would have to be represented as derived classes. Third, new classes need to be added to the ontology every time a new type of service is introduced. A cleaner approach that separates representation of the service definitions from service instances has already been described in Sec. 4.

7.3 Physical Composition

Several standardization proposals aimed at providing infrastructure support to Web service composition have recently emerged including SOAP, WSDL, UDDI, and BPEL. There has also been a lot of interest in the area of dynamic Web service and QoS-based workflow management. Previous efforts in this area like eFlow [4] have investigated dynamic service selection based on user requirements. Zeng et al. [30] propose that the choice of component services in the plan be made at run time for optimality. Instead of making local choices at each step of the composition, the focus is on optimization at a composite level based on a generic QoS model (based on price, duration, reliability etc.) and established linear programming techniques. Other proposals such as METEOR [3] and CrossFlow [10] have considered QoS models for workflows along four dimensions namely time, cost, reliabilty and fidelity. Finally, there has been a considerable effort in the Web service community in identifying the challenges in workflow orchestration between component services. In [13], the authors consider the problem of service composition as a problem of software synthesis where algorithms for matching and composition are based on Structural Synthesis of Programs (SSP) [14]. The SSP language is used as an internal representation language for automated service composition, while DAML-S is used as an external language for describing Web service properties.

8 Conclusion

We have described a two-step methodology for end to end composition of web services by semantically annotating web service components, as well as a prototype that demonstrates this methodology in a domain-specific scenario. Service developers can maintain a registry of web services that goes beyond the traditional UDDI by incorporating semantic descriptions of the components. When a new service requirement arises, it can be expressed in the context of a domain ontology. Our service creation environment can then be used to generate potential workflows for achieving the desired functionality reusing existing web services. This automation of the discovery process results in significant reduction in the time-to-market for the new service.

There are two key innovations in our solution. First, we decouple web service composition into logical and physical composition stages that address complementary integration issues. The first stage focuses on the feasibility of functional composition while the latter deals with efficient execution of the resulting composition. Second, we use scalable and optimizing techniques in each stage that can adapt to changes in the service creation environment.

In the future, we plan to extend the service creation tool to enable more interactions with the developer and provide better support for stitching together message flows between the selected components. We also plan to transition to OWL 1.1 (currently in Beta stage) that provides support for rules. This would enable us to express richer pre-conditions and effects while representing service capabilities. For physical composition, we will continue investigating different instance selection heuristics for QoS matching, not only at the service instance level but also at the level of the composite service. Finally, we want to explore a feedback–based approach where the service creation tool interacts with the service execution infrastructure, to adapt and optimize based on changes in the execution environment.

9 Acknowledgments

The authors would like to thank Anupam Mediratta, Sugata Ghosal, Yigal Hoffner, Christian Facciorusso, Richard Goodwin and Rama Akkiraju for their contributions to the development of the ideas presented in this paper.

References

[1] V. Agarwal, N. Karnik, and A. Kumar. An Information Model for Metering and Accounting. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium*, April 2004.

[2] J. Blythe et al. The Role of Planning in Grid Computing. Proc. of Intl Conference on AI Planning and Scheduling, 2003.

[3] J. Cardoso. *Quality of Service and Semantic Composition of Workflows*. PhD thesis, University of Georgia, 2002.

[4] F. Casati, S. Ilnicki, L. J. Jin, V. Krishnamoorthy, and M. C. Shan. eFlow: A Platform for Developing and Managing Composite e-Services. . Technical Report HPL-2000-36, HP Laboratories, 2000.

[5] G. B. Chafle, S. Chandra, V. Mann, and M. G. Nanda. Decentralized Orchestration of Composite Web Services. In *Proceedings of the 13th International World Wide Web conference*, 2004.

[6] M. B. Do and S. Kambhampati. Sapa: A Scalable Multi-objective Heuristic Metric Temporal Planner. *Journal of AI Research*, 20:155–194, 2003.

[7] P. Doshi, R. Goodwin, R. Akkiraju, and S. Roeder. Parameterized Semantic Matchmaking for Workflow Composition. Technical Report RC23133, March 2004.

[8] K. Erol, J. Hendler, and D. S. Nau. HTN planning: Complexity and expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, Seattle, Washington, USA, 1994.

[9] 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.

[10] J. Klingemann. Controlled flexibility in workflow management. In *Proc. of the 12th International Conference on Advanced Information Systems (CAiSE)*, June 2000.

[11] R. Lara, H. Lausen, S. Arroyo, J. de Bruijn, and D. Fensel. Semantic Web Services: Description Requirements and Current Technologies. In *International Workshop on Electronic Commerce, Agents, and Semantic Web Services*, September 2003.

[12] D. J. Mandell and S. A. McIlraith. Adapting BPEL4WS for the Semantic Web: The Bottom-Up Approach to Web Service Interoperation. In *Proceedings of International Semantic Web Conference*, Oct 2003.

[13] M. Matskin and J. Rao. Value Added Web Services Composition using Automatic Program Synthesis. In *Proceedings of International Workshop on Web Services, E-business, and the Semantic Web*, 2002.

[14] M. Matskin and E. Tyugu. Strategies of Structural Synthesis of Programs and its Extensions. *Computing and Informatics*, 20:1–25, 2001.

[15] S. McIlraith, T. C. Son, and H. Zeng. Semantic Web Services. *IEEE Intelligent Systems, Special Issue on the Semantic Web.*, 16(2):46–53, March/April 2001.

[16] Common Information Model (CIM) Metrics Model, Version 2.7. Distributed Management Task Force, http://www.dmtf.org/standards/documents/CIM/ DSP0141.pdf, June 2003.

[17] M. G. Nanda and N. Karnik. Synchronization Analysis for Decentralizing Composite Web Services. In *Proceedings of the ACM Symposium on Applied Computing*, March 2003.

[18] OWL Services Coalition. OWL-S: Semantic Markup for Web Services. http://www.daml.org/services/ owl-s/1.0/owl-s.html, Nov. 2003.

[19] M. Paolucci, T. Kawmura, T. Payne, and K. Sycara. Semantic Matching of Web Services Capabilities. In *First Int. Semantic Web Conf.*, 2002.

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

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

[22] E. Sirin and B. Parsia. Planning for Semantic Web Services. In *Semantic Web Services Workshop at 3rd International Semantic Web Conference*, 2004.

[23] E. Sirin, B. Parsia, and J. Hendler. Composition-driven Filtering and Selection of Semantic Web Services. In *AAAI Spring Symposium on Semantic Web Services*, March 2004.

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

[25] B. Srivastava, S. Kambhampati, and M. B. Do. Planning the project management way: Efficient planning by effective integration of causal and resource reasoning in RealPlan. *Artif. Intell.*, 131(1-2):73–134, 2001.

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

[27] X. Su and J. Rao. A Survey of Automated Web Service Composition Methods. In *Proceedings of First International Workshop on Semantic Web Services and Web Process Composition*, July 2004.

[28] P. Traverso and M. Pistore. Automated Composition of Semantic Web Services into Executable Processes. In *3rd Int. Semantic Web Conf.,November 9-11, 2004, Hiroshima, Japan.*, 2004.

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

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