Synthy: A System for End to End Composition of Web Services

Vikas Agarwal, Girish Chafle, Koustuv Dasgupta, Neeran Karnik
Arun Kumar, Sumit Mittal, Biplav Srivastava

IBM India Research Laboratory
Block 1, IIT campus, Hauz Khas
New Delhi 110016, India

Abstract

The demand for quickly delivering new applications is increasingly becoming a business imperative today. However, application development is often done in an ad hoc manner resulting in poor reuse of software assets and longer time-to-delivery. Web services have received much interest due to their potential in facilitating seamless business-to-business or enterprise application integration. A web service composition system can help automate the process, from specifying business process functionalities, to developing executable workflows that capture non-functional (e.g. QoS) requirements, to deploying them on a runtime infrastructure. Intuitively, web services can be viewed as software components and the process of web service composition similar to software synthesis. In addition, service composition needs to address the build-time and run-time issues of the integrated application, thereby making it a more challenging and practical problem than software synthesis. However, current solutions based on business web services (using WSDL, BPEL, SOAP etc.) or semantic web services (using ontologies, goal-directed reasoning etc.) are both piecemeal and insufficient. We formulate the web service composition problem and describe the first integrated system for composing web services end to end, i.e., from specification to deployment. The proposed solution is based on a novel two-staged composition approach that addresses the information modeling aspects of web services, provides support for contextual information while composing services, employs efficient decoupling of functional and non-functional requirements, and leads to improved scalability and failure handling. We also present Synthy, a prototype of the service composition system, and demonstrate its effectiveness with the help of an application scenario from the telecom domain.

Key words: semantic web, service composition, application integration, ontology, planning, Quality of Service

Preprint submitted to Elsevier Science
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, 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.

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 and longer time–to–delivery. 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. This could be attributed to the availability of new service providers, new service capabilities, or physical changes in the network or environment – thereby necessitating 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 [1,2]. 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. Further, a web service composition system can enable the end to end automation of business to business and/or enterprise application integration, from the stage of specification to its execution.

To this end, 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\(^1\), composed into flows with a language like BPEL\(^2\), and invoked with the SOAP protocol\(^3\). 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 \([3]\). These approaches by themselves are piecemeal, and insufficient. The former has focused on the execution aspects of composite web services, without much consideration for requirements that drive 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 service composition can be leveraged for application integration, by combining the strengths of both the above approaches. We first formulate the problem of end to end web service composition - from specification to deployment. Next, we present a methodology that, given a formally specified requirement for a new service, stitches together web service components in a BPEL flow that delivers the requirements. In doing so, we identify the key challenges involved in the process of end–to–end composition. One of the challenges stems from the information modeling aspect of web services, that should adhere to the best knowledge engineering practices of conciseness, scalability and manageability. Next, the specification of an executable composed service should comprise of both its control flow (dependence among activities) and data flow (dependence among data manipulations). While planning techniques can be used to generate the control flow, data flows need to be generated by reasoning with the context of input–output parameters of the service. To this end, service composition can be treated similar to the process of software synthesis. Further, service selection is an important step in service composition. When we talk about service selection, we need to efficiently handle functional requirements (represented via the service input–output parameters) and non–functional (QoS etc.) requirements. Finally, a service creation environment should be able to recover from failures that occur during the process of composition. We address each

\(^1\) http://www.w3.org/TR/wsdl
\(^3\) www.w3.org/TR/soap/
of these issues systematically and present a pragmatic end–to–end solution for the problem.

The main contributions of the paper can be summarized as follows:

- A principled two-stage web service composition approach leveraging the differentiation between web service types and their instances. This helps in handling different requirements at each stage, and different means to optimize them. It allows us to achieve scalability and, more importantly, desired level of automation while providing the developer appropriate control over the composition process.

- Synthy, an end to end prototype of the service composition system, that includes: (a) Composition at type level using ontology matchmaking and planning techniques, (b) Composition at instance level that satisfies and/or optimizes non-functional requirements, and (c) Data flow construction for operationalizing the composed service in the form of a BPEL flow.

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 service composition. In Section 3, we formalize the end to end service composition problem and introduce a staged composition approach to solve it. We next describe our approach (Section 4) followed by details on its main aspects – logical composition (Section 5) and physical composition (Section 6). Section 7 illustrates our solution for an application scenario. Section 8 describes how the composition system can handle failures. In Section 9, we provide a summary of related work. Finally, we conclude in Section 10 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 consumer electronics manufacturer.

A customer calls in to report a problem with her electronic item, e.g. a 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 location and resolve the problem (in case of a washing machine, the agent needs to visit the home address provided by the
The service provider would like to create a set of web services that automates 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. Fig. 1 summarizes the workflow in this Helpline scenario.

![Fig. 1. Helpline Service.](image)

Here is a sampling of the component services that may be available in the service provider’s infrastructure: Location tracking, SMS, Call Setup, Customer Database, Agent Database, and 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. Agent Selection.

![Fig. 2. Location-based Agent Selector Service.](image)

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 system 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 needs to formally specify the functional and non-functional requirements of the service to be created. The system 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 end to end composition of the complete flow of Fig. 1 in section 7.

3 Formalizing End to End Service Composition

In this section, we characterize the end to end web service composition problem without actually presenting methods to solve it. The solution is presented in the subsequent sections.

Web service composition is the process of realizing the requirements of a new web service (based on specifications) using the existing component web services. The specifications for web service composition from an end-user can be decomposed into two parts. The first part deals with the desired functionality of the composite service, called the functional requirements. The second part involves specification of the non-functional requirements that relate to issues like performance and availability. The input to an end-to-end web service composition problem is a composite service specification in an appropriate language, e.g., OWL expression for functional requirements, Quality-of-service (QoS) specification for non-functional requirements. We do not restrict the input to any one language, but assume that the specification is done in a manner that would make the subsequent composition feasible.

We differentiate between web service types, which are groupings of similar (in terms of functionality) web services, and the actual web service instances that can be invoked. We believe that the separate representation of web service type definitions from instance definitions helps in handling different
requirements, and different means to optimize them. This, in turn, allows us to work efficiently with large collection of web services. The web service types and instances can be advertised in a registry.

Formally, we denote,

1. \( S = \{S_1, ..., S_\alpha\} \): Set of \( \alpha \) web service types.
2. \( I = \{I_1, ..., I_\beta\} \): Set of \( \beta \) service instances advertised in a registry like UDDI. The mapping of \( S \) into \( I \) is surjective and one-to-many. Assuming each service type \( S_k \) has \( M \) instantiations, \( \beta = M \times \alpha \). When we want to refer to an instance of a particular service type \( S_k \), for convenience, we will also use \( S^{\text{instance}}: \text{S}^{\text{type}} \) without the subscript or instance notation.

**Problem Description:** The *end to end service composition problem* can be stated as follows: Given a set of web service types and the set of instances for each type, along with the specifications of a new service, create an executable workflow that stitches together the desired functionality from the existing services, while satisfying the non–functional requirements.

Note that *workflow* is a set of coordinated tasks. A workflow language like BPEL represents both executable activities (like *invoke*) and structuring activities (like *flow*, *switch*). We will also use the term *plan* from the AI planning literature when referring to automatically generated coordinated tasks. A plan is a sequence of steps, where each step can have concurrent actions. Hence, plan \( P_i = [\text{Step}_1, \text{Step}_2, ..., \text{Step}_{\text{NUMSTEPS}(P_i)}] \) where \( \text{Step}_k \subseteq S \) and \( \text{NUMSTEPS}(P_i) \) returns the number of steps in \( P_i \). We will use \( |P_i| \) to refer to the number of actions in the plan \( P_i \). Each action can be viewed as a software component with a set of input and output parameters. Since a plan can be represented as a workflow, we use the terms plan and workflow interchangeably in the rest of the discussion.

### 3.1 Staged Approach for End to End Service Composition

To solve the end to end service composition problem, we propose a principled two–stage composition approach. The staged approach is designed keeping in mind the best knowledge engineering practices of modularity, conciseness, and scalability, while providing the service developer a fair amount of control to supervise the composition process. Fig. 3 gives an illustration of this staged approach.

---

4 Even though there are no explicit structuring actions in the plan representation, the context of when an action can be executed, can be used to incorporate the same information. Still, a workflow language like BPEL has many additional constructs, like fault handlers, which makes it more general than the plan representation.
As mentioned earlier, this approach distinguishes between service types and instances. The composition first proceeds to generate an abstract plan based on web service types (logical composition), which is subsequently concretized into an executable plan by selecting the appropriate web service instances (physical composition).

The two key elements of the approach are:

1. \( P = \{P_1, \ldots, P_K\} \): A set of \( K \) abstract plans selected after logical composition. Note that an abstract plan \( P_i \) has \( |P_i| \) service types in it. Further, we denote \( \lambda = \max_{1 \leq i \leq K} |P_i| \) i.e. the maximum number of steps in any abstract plan. Then, an abstract plan can be found by searching in \( O(\alpha^\lambda) \) space. For \( K \) plans, the search space is \( O(K.\alpha^\lambda) \).

2. \( W = \{W_1, \ldots, W_L\} \): A set of \( L \) executable plans selected after physical composition. Since each service type has \( M \) instantiations, the search space for selecting an executable plan is \( O(M^\lambda) \) and the search space for selection of \( L \) instantiated plans is \( O(L.M^\lambda) \). Hence, the staged approach can output an executable plan by searching in a total of \( O(K.\alpha^\lambda) + O(L.M^\lambda) \) space.

Note that, a composition approach that does not use the distinction between types and instances can find an executable plan by searching in \( O(L.\beta^\lambda) \) space. The staged approach leads to a significant reduction of the search space to \( O(K.\alpha^\lambda) + O(L.M^\lambda) \) \(^5\).

Further, the mapping of \( P \) into \( W \) is surjective and one-to-many. Recall that the same relationship holds between \( S \) and \( I \). This alludes to the fact that the abstract and executable plans corresponding to the composition can be

\(^5\) For \( \alpha = 10, M = 5, \beta = 5.10 = 50, K = L = 3 \) and \( \lambda = 3 \), we have \( L.\beta^\lambda = 3.50^3 \) while \( K.\alpha^\lambda + L.M^\lambda = 3.10^3 + 3.5^3 \).
considered as (composite) service types and instances, respectively, and reused.

As shown in Fig. 3, the output of the logical composition stage goes to the physical composition stage, the output of which is passed to the execution environment (runtime). Further, we allow multiple workflows to be passed between the stages. The idea of having multiple plans at each stage gives the composition system the ability to pick and choose among them (e.g., based on some optimization criteria) and also provide better failure resiliency (e.g., if an abstract workflow fails to get concretized in the physical composition stage, the system can try a different one). This implies that we need to have the ability to rank the workflows at each stage, based on some criteria. With this in mind, we introduce two ranking functions, $R_{AW}$ and $R_{EW}$, to define a ranking over the set of abstract and executable workflows, respectively. In the remaining part of the section, we shed some light on the nature of these ranking functions.

**Function 1** $R_{AW}(P_i) = f_1(P_i, \Delta_{AW}) \rightarrow \mathbb{R}$. The ranking function is defined over an abstract workflow $P_i$ and a disruption factor $\Delta_{AW}$.

$R_{AW}$ is used to rank a set of abstract plans. The ranking function should incorporate metrics (parameters) that are characteristics of the plan that is generated. For example, the length of the plan, $|P_i|$, i.e. the number of steps in the plan, can serve as a ranking function.

The disruption factor is applicable when a developer supervising the composition process decides to inspect the plan resulting after the logical composition phase. In this context, $\Delta_{AW}$ can be decomposed as $\Delta_C + \Delta_I$. Here, the first disruption factor accounts for change ($C$) in the comprehension of the plan as perceived by the developer. Assume that the developer is familiar with a particular plan. If a new plan is now generated, the user has to again try and understand this plan. This effort is inverse to the similarity between the new ($P_{curr}$) and the old plan ($P_{prev}$). The second disruption factor accounts for any intervention ($I$) by the developer, either by choice or by compulsion, before it is handed off to the physical composition stage. We can estimate $\Delta_C$ and $\Delta_I$ as follows:

- $\Delta_C$: $|P_{curr} - P_{prev}|$
- $\Delta_I$: $\sum_{j=1..\text{NUM STEPS}(P_i)-1} ((\Sigma_{a \in \text{Step}_j} | OUT(a) |) \times (\Sigma_{a' \in \text{Step}_{j+1}} | IN(a') |))$, where $a$ and $a'$ represent actions at consecutive steps of the plan and $IN(a')$ and $OUT(a)$ represent the input and output parameters of their argument actions, respectively. The expression says that the product of the number of output parameters in the predecessor step and the number of inputs in the successor step, aggregated over all the steps, is an estimate of the disruption that can be potentially caused to the developer.

Intuitively, $\Delta_C$ is a measure of the number of actions in the new plan that were
not present in the previous plan. \( \Delta I \) is a measure of the amount of developer’s effort required (from the point of view of software synthesis), to match the input-output parameters between the steps of a particular plan.

One may extend the definition of \( R_{AW} \) to include a factor about how well an abstract plan meets the given specification, thereby allowing partially generated plans to be passed from one stage to another. Finally, in the case where the composition process is fully automated, there is no intervention from the developer, and hence \( \Delta_{AW} = 0 \).

\textbf{Function 2} \( R_{EW}(W_i) = f_2(W_{i}^{QoS*}) \rightarrow \mathbb{R} \). The ranking function is defined over the estimated QoS, with respect to an executable workflow \( W_i \).

\( R_{EW} \) is used to rank among a set of executable workflows. Quality of Service (QoS) is the most common basis to differentiate among workflows. Here, QoS* implies that the measures are obtained from the execution environment and aggregated over periodic intervals. Note that there are different QoS metrics that could be of possible interest for end to end composition (e.g. cost, availability) and we later discuss how to incorporate some of these in our solution.

We next describe an end to end service composition system that realizes the staged approach.

\section*{4 System Overview}

Our end to end service composition system, based on the two-stage composition approach, consists of the following parts:

\begin{enumerate}
  \item \textit{Service Representation:} Representing the available services and their capabilities.
  \item \textit{Requirements Specification:} Specifying the desired functionality of a new service.
  \item \textit{Composition:} Constructing a composition of available services that provides the desired functionality.
  \item \textit{Composite Service Representation:} Representing the new composite service and its capabilities so that it can be programmatically deployed, discovered and invoked.
\end{enumerate}

In a way, the proposed 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 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 two modules:

(1) **Logical Composer**: This module provides functional composition of service types to create new functionality that is currently not available.

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

This basic approach of service creation is illustrated in Fig. 4. 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 abstract plan into an executable workflow that can be deployed and executed, specific instances must be chosen for the component service types in the plan. The Physical Composer uses optimization 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 and performs end to end QoS optimization 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\(^6\), 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 [4,5]. In this paper, we will focus on the Logical and Physical composition stages.

5 Logical Composition

Fig. 5 depicts our system for realizing the Logical Composition stage. Available service types and their capabilities are represented in a Service Capabilities Registry. An Ontology captures the domain model. We use IBM’s SNOBASE\(^7\) 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 service types 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.

---

6 http://www.software.ibm.com/wsdd/zones/was/wpc.html
7 http://www.alphaworks.ibm.com/tech/snobase
5.1 Representation of Service Types

To enable automatic discovery and composition of desired functionality, we need a language to describe the available web services. 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 [6]. Therefore at this stage it suffices to describe the 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)\(^8\) 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 [7] (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). Each instance of Service refers to 0 or more profiles, and 0 or 1 models. In addition, if there’s a model, it must be accompanied by 1 or more groundings. Currently, ProcessModel is the only type of ServiceModel being defined for OWL-S.

OWL-S is designed to describe a single web service instance [7]. This is easily

\(^8\) [http://www.daml.org](http://www.daml.org)
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. [6]. In our present solution prototype, we use the ServiceProfile model of OWL-S to represent web service type definitions. The task of providing descriptions for specific web service instances is deferred to the physical composition phase.

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. 6). 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 an instance will now point to the corresponding ServiceProfileType for mandatory portion common to all instances. It could, however, add its own precondition and effects as discussed later in this section. Similarly, it may support additional outputs as well as inputs, in which case it specifies the default values of additional inputs so that the compositions done using ServiceProfileType remain valid.

ServiceGrounding is a concept that applies to instances rather than types and can stay as it is. A ServiceModel should ideally be encapsulated inside the service interface and not exposed to the external world. Making the model visible outside the service is useful only if it describes the conversational aspect of the web service that would be needed to interoperate with it. Hence, in this case, it should be included in the ServiceType hierarchy since a common conversation model should be applicable to all instances of a service type. In other words, we propose to have an ontology for service types that consists of ServiceProfileType and ServiceModelType. This would be in addition to an ontology for service instances that consists of a ServiceProfile and a ServiceGrounding.

The approach of separating type definitions from instance definitions has been used successfully in data models for distributed systems management [8,9], and has various modeling benefits. A new kind of service can be specified in the ontology by adding an object of type ServiceType, without having to create an actual running instance first. This is not possible in the current OWL-S ontology. Creating an object of ServiceType would include defining the parameters in its profile by populating the ServiceProfileType model, and describing the conversation model by populating the ServiceModelType model. Each actual running instance of this web service would be represented by an object of type Service and include a reference to its ServiceType object. Its ServiceProfile model would contain the service parameters including at least the ones listed in the corresponding ServiceProfileType. Finally, there would
be an instance specific ServiceGrounding defined.

5.1.1 Classifying Types and Instances

Our proposal raises the question of what kind of relationship a web service type has with its various instances. A web service type captures the core functionality of a class of web services. Individual instances belonging to that class of services must adhere to the basic type definition but may be allowed to offer minor variations under some constraints. An important desiderata is that any composition which is produced with the web service type should be still valid when any of its web service instance is selected. This is ensured if the precondition of a web service type is more specific than precondition of some (or more strictly, all) of its instances and its effect is more general than effect of any of its instances. We can summarize the relationship as:

If $S_{\text{instance}}$ is of $S_{\text{type}}$,
1. $S_{\text{type}}$ precondition $\vdash S_{\text{instance}}$ precondition and
2. $S_{\text{type}}$ effect $\dashv S_{\text{instance}}$ effect

The above relationship states that the precondition of the service type entails the precondition of the service instance so that the latter is satisfied whenever the former is. For effects, the reverse is true. With this, given a web service request $R$, when the request matches a service type ($R \bowtie S_{\text{type}}$), the relationship between $R$ and the web service instances would be:

1. $R_{\text{precondition}} \bowtie S_{\text{precondition}}$ $\Rightarrow (\forall S_{\text{instance}} : S_{\text{type}}) R_{\text{precondition}} \bowtie S_{\text{instance}}$ precondition
   and
2. $R_{\text{effect}} \bowtie S_{\text{effect}}$ $\Rightarrow (\forall S_{\text{instance}} : S_{\text{type}}) R_{\text{effect}} \bowtie S_{\text{instance}}$

According to it, if a request $R$ matches a web service type, where matching can be exact or defined over a range as in [10], all the instances of the web service type will also match. While this relationship would guarantee that
Compositions are valid when the abstract plan is concretized, it can be overly restrictive because the precondition of the web service type is required to be more specific than all its instances. We will call this as the \textit{strict} relation. To relax the restriction, we use the insight that eventually each web service type referred in the abstract plan will be instantiated by only one web service instance. Therefore, as long as we could guarantee that if a request $R$ matches a web service type, some at least one instance of the web service type will also match, the abstract will be successfully concretized and the composition will succeed. That is,

\begin{enumerate}
\item $R_{\text{precondition}} \triangleright S^{type}_{\text{precondition}} \Rightarrow (\exists S^{instance-i}_{\text{precondition}}): S^{type}_{\text{precondition}}$ and
\item $R_{\text{effect}} \triangleright S^{type}_{\text{effect}} \Rightarrow (\exists S^{instance-j}_{\text{effect}}): S^{type}_{\text{effect}}$ and
\item instance-$i$ = instance-$j$
\end{enumerate}

The decision of whether to follow the \textit{strict} or \textit{relaxed} relationship during domain modeling is one of balancing tradeoffs. With the former, the abstract plans can be automatically concretized because all its service instances are guaranteed to preserve composition. With the latter, a service type has to only be more specific than at least one of its instances and this would simplify building of the services ontology (e.g., more instances for a type). During the concretization of abstract plan, all instances might need to be explored for say, optimality. In this case, additional constraints will have to be checked for instances whose preconditions are more specific than that of their type. Checking these additional constraints may require the intervention of a developer.

We adopt the \textit{relaxed} relation above as the guideline for our domain modeling. In the absence of the model differentiating service types from instances in OWL-S, we use \textit{ServiceProfile} model to represent web service type definitions in our service composition system [11]. The task of providing descriptions for specific web service instances is done using the Web Services Matchmaking Engine (WSME) [12] as discussed later in Physical Composition.

\subsection{5.1.2 Modeling Non Functional Service Capabilities}

The functional capability (FC) of a web service describes its core functionality. It is expressed through IOPEs that capture the transformation performed by this service. The non-functional capabilities (NFCs), on the other hand, help in characterizing the service further by capturing its optional features, such as QoS etc. OWL-S has provision to represent NFCs through profile attributes.
which may contain parameters other than the functional IOPEs.

Since NFCs inherently capture properties of service instances (and not of types), they are not needed during functional composition. In contrast, FCs form the core of the functional composition process. NFCs play an important role during selection of appropriate service instances in order to meet the end-user requirements. The current OWL-S only deals with service instances and therefore all the functional as well as non-functional attributes are in the ServiceProfile. In our proposed OWL-S upper ontology (presented in Fig. 6), the FCs get represented in ServiceProfileType. The ServiceProfile of an instance inherits these FCs from the ServiceProfileType and adds the NFCs to it.

In some domains it may be desirable to model certain service features as mandatory for all instances of a service type. In military applications, for example, it may be necessary to make all service instances secure. For such domains, it seems logical to model NFCs such as security in the service type itself. These non-functional capabilities now form a part of the core functionality. They are included in ServiceProfileTypes and are utilized in selecting service types during the logical composition phase.

In our system, the instance-specific NFCs are stored using the Web Services Matchmaking Engine (WSME) [12] as discussed later in physical composition. NFCs at the service type level can be added using OWL-S.

5.2 Requirements Specification

In order to create a new service, the developer should describe the required functionality as well as non-functional requirements such as availability, response time, cost etc. We use OWL-S for representing the functional requirements (in IOPE terms) of the composite service. The developer is presented with a graphical user interface to specify these requirements. The system 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 system processes the requirements incrementally. The preconditions and effects are logical terms and expressions, 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

9 Services can be matched using arbitrary expressions as defined with OWL-S 1.1 but for planning, we use a state representation consistent with the STRIPS assumptions[13].
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 [14]. Exploring the feasibility of metric planning for end to end web service composition would be an interesting area for future work.

For the LAS composite service, the precondition (or the initial state of the composition problem) asserts that CustomerLocation is known and the effect (or the goal state) is to find the agent (AgentID) nearest to the customer location.

5.3 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 [13] \( P \) is a 3-tuple \( < I, G, A > \) 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\(^{10}\). A state \( T \) is simply a collection of facts with the semantics that information corresponding to the predicates in the state holds (is true). An action \( A_i \) is applicable in a state \( T \) if its precondition is satisfied in \( T \) and the resulting state \( T' \) is obtained by incorporating the effects of \( A_i \). 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 \( G \)\(^{11}\). A planner finds plans by evaluating actions and searching in the space of possible world states or the space of partial plans. While planning is known to be a hard problem from various computational complexity studies[15,16], it can be very efficiently solved in practice. 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 [3].

For our service creation system, planning for web services has some unique characteristics (refer to Fig. 5):

- The nature of planning is limited contingency planning (CP). 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,

\(^{10}\) A more restricted planning definition calls for \( G \) to be completely specified so that the generated plan has no unspecified side-effects.

\(^{11}\) A plan can contain none, one or more than one instance of an action \( A_i \) in \( A \).
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 unimportant or unlikely ones, the user may manually insert a default branch.

For such a situation, we have developed a novel user-driven search-control methodology which takes input from user and then uses them to efficiently focus the search. These inputs are: conditions of interest, the type of plan desired and the number of conditions to handle. Our approach employs user-inputs on the AND-OR graph of CP’s belief states to prune space in the AND part of the graph and additionally uses the well known planning-graph (PG) based heuristics to do the same in the OR part. The approach efficiently finds contingent plans focusing on user interest. This is complementary to recent utility based methods for contingency selection. We have implemented such a contingent planner in the Planner4J framework [17] and its details can be found in [18].

• Since the number of service types can be large, Filtering is needed to remove irrelevant web services. 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 (at least one effect unifies with a goal) or to the preconditions of any service which can potentially contribute to the goal.

• Our Matchmaker [19] matches the preconditions of a web service with the effects of another up front during filtering. This is in contrast to typical matchmaking of web services where the preconditions and effects of the request are matched respectively with the same fields of the advertisements[10]. We also envisage the matchmaking to happen up front before planning. Another approach can be to perform matchmaking as needed during planning [20]. This can support more expressive matching (e.g. involving expressions of initially unknown terms that are evaluated at
runtime) but at the cost of slower performance due to frequent reference to the ontology unifier.

- We can provide partially complete 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. But this raises the question about which incomplete plans should be returned from the large set of possible plan fragments. 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 and is returned. This planner feature is especially valuable when the ontology development has not stabilized.

Scalability and Performance: With the listed features, planning is scalable, efficient and user friendly. We have reported empirical analysis of the planner in standard planning domains in [18] and only illustrate some results here. All results were taken on a IBM ThinkPad which has 1.6GHz Pentium 4 CPU and 512 MB of RAM running Red Hat Linux V9 on it. 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 and gives the plan(s) in milliseconds. And without any user input about branch preferences, the planner is still comparable with existing contingent planners. We believe that these results will carry over to the current web service domain as well.

In Fig. 7, the planner was invoked for the LAS service. Recall that 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 Logical Composer is a 4-step plan that can accomplish the goal. The created service is added to the service capabilities registry by the user.

5.4 The Abstract Workflow

The planner generates plan which can have sequence, choice and concurrency among actions. In future, we will explore generating plans with loops. The plan representation is a sequence of steps, where each step may have concurrent actions (recall the discussion in Section 3). Each action corresponds to a web service type in our case. 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. Note that 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.

6 Physical Composition

![Fig. 8. Physical Composition.](image)

The Physical Composer views the abstract workflow as a template [21] for the composite service, which in conjunction with the non–functional requirement specifications, drives the process of matching each service type to a corresponding service instance. Note that this is a non–trivial problem and involves a number of issues related to QoS optimization, satisfying non-functional requirements (constraints), data flow orchestration, data type matching and invocation protocol matching. While some of these issues can be resolved in an automated manner, others might require manual intervention from a developer supervising the composition process.

The Physical Composition stage proceeds in steps, illustrated in Fig. 8. The first step deals with representation of the available service instances as well as the non-functional requirements associated with the composite service. Next, a set of matching instances is selected by the Matchmaker for each service type in the abstract workflow. This step also filters out the instances based on local non–functional requirements. Next, the Instance Selector picks exactly one instance from the set of matching instances for each service type. The instance selection process takes into account global optimization criteria (e.g. QoS) as
well as end-to-end constraints that are specified as part of the non-functional requirements. We elaborate on these optimization criteria and local/end-to-end constraints later in the section. Finally, we deal with the issues of data flow orchestration and data type matching in order to generate an executable workflow. We now describe each of the above steps in greater detail.

6.1 Representation of Service Instances and Requirements

As in logical composition, we require a representation for service instances and requirements to facilitate physical composition. It has been established that directory services, such as UDDI, are important but insufficient for this purpose [12]. Specifically, one needs to add 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) [12] – an engine capable of matching complex entities, and a Data Dictionary tool for defining the language for the matching process. Matching is performed between a set of advertised service instances and requirements specified by the consumer. In this case, the requirements come from the abstract workflow (i.e. types that need to be concretized) and additional matching criteria (i.e. non-functional constraints like QoS) that are 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 for a service is submitted by the provider to WSME and is persistent, 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.

We illustrate the Data Dictionary definitions used for composing the Helpline Service in Fig. 9. 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
Fig. 9. Advertisement and Query Formats for Helpline Service.

type, method name and WSDL information, along with QoS-specific metrics like (expected) response time, availability 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.

Non-functional requirements like QoS constraints from the developer are specified in a flat file. These constraints can be end–to–end, for example the composite LAS service should have availability more than 0.8, cost should be less than $375, response time should be less than 2 minutes etc. Such requirements can also be on a component basis - LocationTracker service in LAS should have cost less than $100. As discussed previously, our model is agnostic to the particular language in which the requirements are specified. We are exploring the use of sophisticated QoS specification languages like WSLA \(^{12}\) in the future for this purpose.

6.2 Matchmaking

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: \(\text{return}(\text{my.MaxCost} \geq \text{your.cost})\). 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; the Data Dictionary contains those definitions.

\(^{12}\) http://www.research.ibm.com/wsla
A Query 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 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 [12].

In our case, we would like to know all the instances available to us for each action in the abstract workflow. Given a set of instances in the service registry, the function of Matchmaker is to match and return the instances corresponding to each service type given by the abstract workflow. In addition, this component filters the instances matched based on the requirements imposed at the service type level. These requirements will be specified by the user based on his domain knowledge and expectations. Consider our LAS example - if the requirement for LocationTracker is to cost less than $100, we can specify my.MaxCost as $100 for this service type and use it to query WSME for matching instances. The result would be all instances of LocationTracker having cost less than $100.

Other techniques, such as those proposed by [22,23], can also be used alternatively to tackle the problem of instance selection for a service type based on the local optimization criteria. It is important to note, however, that this process matches individual instances, but does not take into account the end to end QoS requirements of the composite service. For example, the requirement for the composite LAS service could be to have a total cost of less than $375. In order to achieve end-to-end QoS satisfaction, we need to select the individual service instances for each action in the workflow in such a manner that the aggregate non functional capabilities of the instances selected satisfy the composite service requirement. We discuss our approach to this problem in the next section.
6.3 Satisfying End-to-End QoS Requirements

The **Instance Selector** gets an abstract plan (a DAG of service types) from the Logical Composer, an optimization criteria specified by the user, and a set of instances from the **Matchmaker** for each service type in the abstract plan. It then selects an instance for each service type based on the specified optimization criteria. Our solution framework is based on the fact that the end-to-end QoS of the composite service is determined by the individual QoS metrics of its component services. It is built on the *multi-dimensional QoS* model and the mathematical formulation discussed in [24,25]. We give only a brief summary of the model and the formulation in this section. Interested readers should refer to the original papers for more details.

We consider only *cost* \((C,\) the fee that a service requester has to pay for invoking an operation on the service), *response time* \((R,\) the expected delay between the moment when a request is sent and the moment when the results are received) and *availability* \((A,\) the probability that the service is accessible) in the QoS model. We feel that these three are the most generic and most frequently used parameters in service selection by users. However, the model is generic to capture any number of QoS dimensions.

The basic idea is to optimize a normalized function that captures the end-to-end values of each of the QoS dimensions according to some relative weights. We use the Simple Additive Weighting (SAW) technique [26] to obtain the normalized objective function based on the aggregated values of the quality dimensions. The relative weights are provided by the user and reflects the importance of each of the quality dimension with respect to the other. The QoS values for each of the service instance are obtained from the data dictionary tool (refer Fig. 9) in WSME. These can also be obtained from a monitoring infrastructure and computed using the QoS estimation formulae presented in [22].

The aggregation functions for QoS dimensions are defined for an *executable plan*, where an executable plan represents one particular execution of the composite service. An executable plan is derived from an abstract plan by grounding each service type to an instance. Simple summation is used to compute aggregate end-to-end value of cost, \(Q_C\), for one particular executable plan of the composite service. Similarly, multiplication is used for availability \((Q_A)\) and summation over *critical path* is used for response time \((Q_R)\). Other aggregation techniques like the ones defined in [22] can also be used here.

Having defined the aggregation functions we now proceed to compute the normalized values for the QoS dimensions.
Normalization

Let,

\[ V_C = \begin{cases} 
\frac{(Q^\text{max}_C - Q_C)}{(Q^\text{max}_C - Q^\text{min}_C)} & \text{if } Q^\text{max}_C - Q^\text{min}_C \neq 0 \\
1 & \text{if } Q^\text{max}_C - Q^\text{min}_C = 0 
\end{cases} \]  

(1)

\[ V_R = \begin{cases} 
\frac{(Q^\text{max}_R - Q_R)}{(Q^\text{max}_R - Q^\text{min}_R)} & \text{if } Q^\text{max}_R - Q^\text{min}_R \neq 0 \\
1 & \text{if } Q^\text{max}_R - Q^\text{min}_R = 0 
\end{cases} \]  

(2)

\[ V_A = \begin{cases} 
\frac{(Q^\text{'A} - Q^\text{min}_A)}{(Q^\text{max}_A - Q^\text{min}_A)} & \text{if } Q^\text{max}_A - Q^\text{min}_A \neq 0 \\
1 & \text{if } Q^\text{max}_A - Q^\text{min}_A = 0 
\end{cases} \]  

(3)

where,

\[ Q^\text{'A} = \ln Q_A \]

\( Q^\text{max}_C \) and \( Q^\text{min}_C \) are the maximal and minimal values of \( Q_C \) (aggregated value of cost for an executable plan). Similar terms are defined for response time and availability. The maximal and minimal values can be computed without iterating over all the executable plans. For example, in order to compute \( Q^\text{max}_C \), we can select the most expensive instance for each service type and sum up all the execution costs.

Since, the aggregation function for availability is not a linear function we linearize it using a logarithm function. Faulty instances with availability of 0 are filtered out before applying the aggregation functions. We now summarize the IP formulation of the problem in the following section.

Objective Function

Maximise: \[ W_C \cdot V_C + W_R \cdot V_R + W_A \cdot V_A \]  

(4)

\( W_C, W_R, W_A \in [0, 1] \) and \( \sum_j W_j = 1 \). \( W_j \) represents the weight of the QoS dimension \( j \).

Constraints

(1) Task Assignment: The abstract plan contains a set of types \( T \). For each service type \( j \) in \( T \), we can select a service instance from the set of instances \( S_j \) that can be assigned to it. Let \( y_{ji} \) be an integer variable,
such that $y_{ji}$ is 1 if service $i$ is selected for task $j$, 0 otherwise. Given that $y_{ji}$ denotes the selection of instance $i$ for type $j$ the following constraints must be satisfied.

$$\sum_{j \in T} y_{ji} = 1, \forall i \in S_j$$  \hspace{1cm} (5)

(2) Processing Time: Let $R_{ji}$ denotes the response time of instance $i$ when assigned to task $j$, and $w_j$ denotes the expected duration of task $j$ knowing which instance was assigned to it. Then we have the following constraint:

$$\sum_{i \in S_j} R_{ji} \cdot y_{ji} = w_j, \forall j \in T$$  \hspace{1cm} (6)

(3) Task Precedence: Let $x_k$ denote the expected start time of task $k$. Also, let $j \rightarrow k$ denote that task $j$ is predecessor of task $k$. Then we have the following constraint:

$$x_k \geq w_j + x_j, \forall j \rightarrow k, \hspace{0.5cm} j, k \in T$$  \hspace{1cm} (7)

(4) Make Span: For all tasks $j$, the end to end response time of the composite web service must be greater than or equal to the sum of start time of task $j$ and the expected duration of task $j$.

$$Q_R \geq w_j + x_j, \forall j \in T$$  \hspace{1cm} (8)

(5) Dominance: Let $z_{ji}$ be an integer variable that has value 1 or 0: 1 indicates that the instance $i$ selected is critical (assigned to a type $j$ which is on the critical path) and 0 otherwise. The variable $z_{ji}$ will always be less than or equal to assignment variable $y_{ji}$.

$$z_{ji} \leq y_{ji}, \forall i \in S_j, \forall j \in T$$  \hspace{1cm} (9)

(6) Total Cost: The end to end cost of the composite web service is equal to the sum of the costs of all instances $i$ selected for type $j$.

$$Q_C = \sum_{j \in T} \sum_{i \in S_j} C_{ji} \cdot y_{ji}$$  \hspace{1cm} (10)

(7) Total Availability: The total availability of the composite web service is equal to the product of availability of all critical instances $i$ selected for type $j$.

$$Q'_A = \sum_{j \in T} \sum_{i \in S_j} (A'_{ji}) \cdot z_{ji}$$  \hspace{1cm} (11)

where,

$$A'_{ji} = \ln A_{ji}$$
Total Response Time: The total response time of the composite web service is equal to the sum of response times of all critical instances $i$ selected for type $j$.

$$Q_R = \sum_{j \in T} \sum_{i \in S_j} R_{ji} \cdot z_{ji}$$  \hspace{1cm} (12)

Solving equation 4 subject to constraints 5, 6, 7, 8, 9, 10, 11, 12 suggests the need for an IP solver. It can be shown that the solution of these equations will always result in an integral optimal solution [24]. Interesting extensions of the above problem occur when we add practical user constraints like budget, availability, total response time etc. These can be easily incorporated into the formulation by adding a constraint for each. For budget and response time and availability these will be as follows:

**Budget:** $Q_C \leq T_C$  \hspace{1cm} (13)

**Response Time:** $Q_R \leq T_R$  \hspace{1cm} (14)

**Availability:** $Q_A \geq T_A$  \hspace{1cm} (15)

where, $T_C$ is the user’s budget, $T_R$ is the total required response time, and $T_A$ is the availability of the composite service. These extensions are important practical problems since the user is typically interested in providing end-to-end service within certain constraints. However, solving it is a non trivial task. The problem can be reduced to the knapsack problem [24] and hence NP-hard [27]. Solving it with LP relaxation may not necessarily yield an integer solution. Hence, heuristics based solutions are required to be able to solve this problem (e.g. convert a fractional solution obtained by LP relaxation to an integer solution).

Multiple abstract plans can be passed by the Logical Composer to the Physical Composer. In this case we need a way to rank the solutions obtained by Instance Selector for each plan. Towards this end, the value of the objective function 4 obtained by solving the optimization problem can be used as a measure of the quality of binding for that particular plan. We show how this ranking can be used for failure handling in Section 8.

6.3.1 Example

Fig. 10 lists the various types and instances available for the LAS example discussed earlier to demonstrate the instance selection part of our system. Here cost is in cents, response time in milliseconds and availability in fraction.
<table>
<thead>
<tr>
<th>QoS</th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GetAgentMobileNumbers</td>
<td>LocationTracker</td>
<td>DistanceCalculator</td>
<td>OptimalAgentSelector</td>
</tr>
<tr>
<td>$S_{11}$</td>
<td>$S_{12}$</td>
<td>$S_{13}$</td>
<td>$S_{21}$</td>
<td>$S_{22}$</td>
</tr>
<tr>
<td>C</td>
<td>300</td>
<td>150</td>
<td>200</td>
<td>120</td>
</tr>
<tr>
<td>R</td>
<td>3</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>A</td>
<td>0.9</td>
<td>0.95</td>
<td>0.9</td>
<td>0.85</td>
</tr>
</tbody>
</table>

Fig. 10. Instances for Tasks in LAS

<table>
<thead>
<tr>
<th>Instances Selected</th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>without budget constraints</td>
<td>$S_{13}$</td>
<td>$S_{21}$</td>
<td>$S_{33}$</td>
<td>$S_{41}$</td>
</tr>
<tr>
<td>with budget constraint</td>
<td>$S_{12}$</td>
<td>$S_{22}$</td>
<td>$S_{32}$</td>
<td>$S_{41}$</td>
</tr>
</tbody>
</table>

Fig. 11. Instance Selection for LAS

There are total of 4 types and maximum of 3 instances for each task. Thus we have:

$$T = \{T_1, T_2, T_3, T_4\}$$

Let $S_{ji}$ represent instance $i$ selected for task $j$ then we have

$$S_1 = \{S_{11}, S_{12}, S_{13}\}$$

$$S_2 = \{S_{21}, S_{22}, S_{23}\}$$

$$S_3 = \{S_{31}, S_{32}, S_{33}\}$$

$$S_4 = \{S_{41}, S_{42}, S_{43}\}$$

We solved the optimization problem without any user constraints and with a budget constraint (budget of 375). The solution is summarized in Fig. 11. As can be seen, the solver picks up different instances (for types $T_1$, $T_2$, and $T_3$) to satisfy a given set of constraint and at the same time maximize the objective function. A sample BPEL code indicating the selection of instances without budget constraint is shown in Fig. 13. We use the notation $typename + \langle i \rangle$, where $i$ is the instance selected, to indicate the invocation of the service type.

In our system we use the CPLEX solver\(^\text{13}\) from ILOG to solve these equations. For the LAS example we got an integral solution with and without budget constraint. However, this will not always be the case and we wish to investigate heuristics based approaches for solving the instance selection problem in its full generality.

\(^\text{13}\)http://www.ilog.com/products/cplex/
6.4  BPEL Generation

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\textsuperscript{14}. 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.

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 following few subsections, we discuss this issue in more detail and present pragmatic solutions for generation of data flow.

6.4.1  Problem of Data Flow Construction

When we seek to operationalize composed plans, we are in fact generating programs. A program contains the specification of both its control flow (the dependence among activities) and the data flow (the dependence among data manipulations). One of the main differences between knowledge engineering and programming, as described in [28]\textsuperscript{15}, is that while logic sentences in the former tend to be self-contained, the statements in a program depend heavily on previous ones.

\textsuperscript{14}http://www.eclipse.org/emf
\textsuperscript{15}Chapter 8, Page 222
on surrounding context. Planning techniques can be used to easily generate the control flow for the composite service given the precondition and effect information for available service types, but generating the complete data flow needs reasoning with contexts of inputs and outputs. For the full composition, data flow has to be produced between dependent services to make the plan executable.

In Fig. 2, the LAS Service accepts two sets of locations - one of these sets is that of the customer and the other corresponds to the list of agent locations. Even if the distinction of semantics of these two sets is not necessary for generating the control flow, it could be important for the data flow. Specifically, they can have different meanings and different data (message) types. In the LAS scenario, to determine the relation between input/output of component services, we must figure out things such as the following:

1. customer locations (obtained from ProblemTicket) go directly to the DistanceCalculator service.
2. distinguish the semantics of the two sets of locations coming as input to the DistanceCalculator service.
3. distinguish the semantics of the locations coming from the customer as corresponding to her home or office address.

To operationalize the workflow of the composite service, we need support for incorporating context with the IO parameters of component web services. One option is to introduce specific terms in the domain ontology, one term for each possible concept and each valid context. However, this makes the ontology large and brittle. This consequence is well understood in knowledge engineering [28] and that is the reason very specific terms are not recommended in an ontology.

In programming languages the issue of data flow is resolved by specifying an ordering among the parameters of a function or procedure. A human developer could then look at the language specification and specify the parameters accordingly. However, in the web service composition scenario, software programs cannot automatically derive and interpret semantics of all parameters just from the available ordering. The context for the inputs and outputs need to be made explicit.

The semantics of each input/output parameter can be expressed along two dimensions. The first one specifies the meaning of the parameter as intended by the service designer. For instance, the designer of Distance Calculator Service could designate one LocationList parameter as the Customer addresses and the other one as the locations of Agents. The second dimension is dictated by the composition of which this service becomes a component. If in a composition, the input parameter Name to a directory service is assigned the
label Customer, the output Location should be assigned the same label.

### 6.4.2 Context Resolution using Roles

We seek to solve the problem of context resolution by explicitly encoding the context for inputs and outputs using the notion of roles. A role is a term that qualifies a concept. That is, for any concept $\varphi$, $(\psi \varphi)$ specifies that the role played by $\varphi$ is $\psi$. Roles are optionally specified on the input and output parameters by the service developer. They come from a separate ontology, and are structured and standardized in a domain similar to concepts. Fig. 12 shows a sample role ontology for roles that could be played, like transfer and expertise roles. Depending on need, a parameter can have either one, multiple or no specific roles. In Fig. 2, the user assigns the roles of Customer and Agents to the two input addresses for the DistanceCalculator service.

![Role Ontology](image)

**Fig. 12. Sample Role Ontology**

In [29], the authors give an extensive coverage of how context is handled in knowledge representation in AI. Their solution is to explicitly model context as a resource and they introduce terms to specify lifting rules so that propositions could be generalized across contexts to serve their data aggregation application. In comparison to the roles, the context of [29] means that if $\text{ist}(c_i, \varphi)$, the proposition $\varphi$ is true in context $c_i$. The two usages can be combined - for example, $\text{ist}(c_i, \psi \varphi)$ means that the proposition $\varphi$ has the role $\psi$ in the context $c_i$. Currently OWL-S does not support the notion of roles for service representation.

A key motivation for defining roles is that they should be generic in nature. If a role can be attached with multiple concepts, it reduces brittleness in the ontology by eliminating the need for specific terms. Introduction of roles with the concepts also presents the user with the task of defining the roles played by the input and output parameters in the specification of the desired service. A tool can help the user look up the various role combinations accepted by
existing concepts. We are currently in the process of building a role ontology for the telecom domain.

### 6.4.3 Data Flow Construction using Roles

Roles can be propagated so that input or output or both can be associated with new roles in the presence of roles coming from requirement specification and/or those of other services. New roles can be acquired while matching a specification with a service instance or from the input to the output of a service and vice-versa. However, there cannot be generic rules for role propagation as it should depend on the way the service processes its inputs to generate outputs. Please refer to [30] for a more elaborate example and detailed discussion.

Assigning roles has two benefits - on the one hand, role disambiguates between multiple instances of the same concept in a service profile thus clarifying the intended usage of the concept in the service. On the other hand, it enables the creation of a context using which the data flow from other service to this service and vice-versa can be constructed. In Fig. 2 for example, using role alignment (matching), we can disambiguate between the locations arriving from the customer and those corresponding to the field agents. Association of roles with parameters of a web service also provides an extra dimension for matching requirements. A match-making tool would try to search services for which the input parameters have roles that fit the description of the requirement.

After the data flow has been constructed, the BPEL generated might still 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. 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.

Fig. 13 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 Section 2. Further, once physical composition is done, the WSDL/BPEL description of this new service is added to the registry, and can be later used in the composition of some other service.
Fig. 13. BPEL code for the LAS service.

7 Composing the Helpline Service

We have developed Synthy, a working prototype of the end to end composition system. We next discuss how Synthy 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 Synthy. The user can then add this service to the registry so that it is available for reuse.

For developing the Helpline service, the user may choose to use Synthy to explore basic services available, build appropriate composite services, and finally build the Helpline service. Alternatively, the user could ask Synthy to build the Helpline service at the outset using the available services, and let it search through the set of possible plans. We expect the user to prefer the
Fig. 14. Specifying input in the Composition Tool

Fig. 15. Logical Plan for Helpline Service
former approach, when the scenario is large and the user wants to control the composition.

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, Synthy is invoked for the overall Helpline service with a precondition of `ProblemHTMLForm`, and the effect of `ProblemResolutionStatus` as shown in Fig. 14(a). The Logical Composer produces the plan shown in Fig. 15(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\(^\text{16}\). Finally, the Physical Composer takes the abstract workflow and generates the appropriate BPEL (similar to Fig. 13).

**Deployment:** The BPEL code is then deployed onto an execution environment which can orchestrate the composite service in a centralized or decentralized manner. The execution environment used in Synthy is based on decentralized orchestration\([5,4,32,33]\). However, our service creation system is agnostic to it and can use either. In decentralized orchestration, a composite service is broken into a set of partitions (called topology), one partition per component web service. The partitions are collocated with the web service. Each partition acts like a proxy that processes, transforms and manages all the incoming and outgoing data at the component web service as per the requirements of the composite service. The partitions execute independently and interact directly with each other using asynchronous messaging without any centralized control. Decentralized orchestration yields performance benefits by exploiting concurrency and reducing the data on the network.

8 Using Multiple Plans to Handle Failures

Failures can occur at various stages of composition. At the logical composition stage no feasible plan may exist for the given specification of the composite service, whereas at the physical composition stage no feasible bindings may exists for the abstract workflow generated in the previous stage. One of the challenges of a service composition system is to take proper corrective actions when such failures occur. While some failures can be taken care of automatically, others might require the intervention of the service developer.

A common way to handle failure is by having redundancy in the plans which are passed across stages. Specifically, in the staged approach, we allow \(K\)

\(^{16}\)Designing heuristic functions so that user intent is respected is an active area of research in Hierarchical Task Network planning \([31]\).
abstract workflows to be passed from the Logical Composer to the Physical Composer. During instance selection, we can then apply our matchmaking and QoS-based optimization techniques to generate $L \leq K$ executable workflows. The executable (BPEL) flows can be finally passed to the execution infrastructure. The availability of multiple workflows at different stages allows the composition system to switch to a different workflow, in case the process fails with a particular choice. However, with multiple workflows being passed between the stages, we need to have a mechanism to rank the workflows that are generated at these stages. With a ranked list of workflows in place, the composition system proceeds with the highest ranked workflow and in the case of a failure switches to the next workflow in the list, and so on. As discussed earlier, we can define ranking functions $R_{AW}$ and $R_{EW}$ over the set of abstract and executable workflows, respectively. In the current system, we use the length of the abstract plan, i.e. the number of actions in the plan, as the function $R_{AW}$. Similarly, in the physical composition stage, we use the quality of the binding (as discussed in section 6) as the function $R_{EW}$. At the time, we are extending the ranking functions to incorporate other factors like comprehension, developer intervention and additional QoS metrics.

During logical composition different kinds of failures may occur. The planner may not be able to find any plan which can satisfy the specification of the composite service. In this case, an exception is generated for the developer indicating that the current set of services in the repository are insufficient to create the specified composite service. The planner can also give a partial plan that is closest to the goal state (specification). This helps the developer in visualizing what can be composed with existing services and what are the missing functionalities that need to be developed. Also, the Logical Composer can generate multiple abstract plans if they exist and feed it to the Physical Composer. This is helpful in case an abstract plan could not get concretized due to failures in the physical composition stage. In this case, the Physical Composer chooses a different plan without going through the logical composition stage again.

During physical composition two kinds of failures can occur – a binding may not exist for a given service type in the abstract workflow or the QoS requirements specified may not be satisfied. If a binding does not exist for a service type in the abstract plan, then the Physical Composer can try using a different plan that (possibly) does not use the particular service type. If no such plan exists, the Composer uses the intervention of the developer to go back to the logical stage and regenerate an abstract plan, without using that service type. Similarly, in case the QoS requirements are not satisfied, the developer could be required to relax some of the QoS constraints that could not be satisfied. Finally, the Physical Composer can generate multiple executable workflows (ranked based on QoS metrics) and feed it to the runtime engine. This can be helpful in scenarios where a workflow could not get executed due
to runtime failures. In this case, another BPEL flow can be deployed without going through the Physical Composer.

Not much research has happened on dealing with failure of the composition process. Situations where available services only cover a part of the range of the input type have been handled in [34].

9 Discussion and Related Work

We have described a two–step approach for end to end composition of web services by semantically annotating web service components, as well as a prototype that demonstrates this approach in a domain-specific scenario. The two staged approach makes our end-to-end composition approach more scalable because the first stage has to deal only with service types and the second stage has to deal with instances of only those types selected in the first stage. In the logical composition stage we use a Filter to prune the set of candidate services before actually composing the abstract workflow. This helps in reducing the complexity of the search process. Further, the planner that we use is also scalable and efficient as discussed in Section 5. Moreover, the IP solver used in the physical stage has minimal overheads and is scalable as shown by the experimental results of [25]. The staged approach also helps in better failure localization and recovery because failure resolution is attempted in stages starting from the most frequent factors for composition disruption: it is initiated at the execution stage and then moves to the physical stage and eventually to logical stage.

The literature on web service composition is extensive, consisting of promising results and many challenges [35,36,2]. To put it in perspective of this paper, we organize this section into three subsections. The first relates to design approaches for end to end composition. The second and third subsections discuss work related to logical composition and physical composition, respectively.

9.1 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 [37]. 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 [38] 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 [39], 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 comparison to our approach, Mandell and McIlraith [40] extend a BPEL engine to support runtime service selection using a semantic discovery server. They present an integrated technique for automating customized, dynamic binding of Web services together with interoperation through semantic translation.

[41] argues that web services composition cannot be seen as a one-shot plan synthesis problem defined with explicit goals but rather as a continual process of manipulating complex workflows, which requires to solve synthesis, execution, optimization, and maintenance problems as goals get incrementally refined. A formalization and evaluation of planning-based approaches for end to end composition and execution of web services is presented in [42]. It classifies approaches based upon the amount of freedom given to the user to intervene and the kind of input they take. The analysis shows the benefits of a staged composition approach on the most important metrics for application integration.

Aggarwal et al. [21] present a tool to bind Web services to an abstract process, based on business and process constraints and to generate an executable process. Their abstract process is similar to our notion of abstract plan. Also, they use service template to represent the functionality of services and to capture the QoS attributes. We use service types to provide the functionality of services, while rendering the flexibility of provisioning QoS attributes both at the type and the instance level. Furthermore, their work involves manually creating abstract processes containing templates. On the other hand, we present an end–to–end approach for automatic composition of web services, starting from specification of requirements to generation of executable processes.

Also, METEOR-S uses the notion of a service template as compared to our
service type. A service template refers to the description of a single web service which consists of a set of operations with their inputs and outputs\[43\]. A service type is not very different in spirit since a type is nothing but a behavioural template for its instances \[44\]. However, using OWL-S for describing the behaviour enables us to represent preconditions and effects of a service in addition to inputs and outputs. This allows functional composition to happen before the actual data types of input/outputs can be matched for creating the data flow. In fact, even though the terms for input and output come from an ontology, having same inputs and outputs does not guarantee that the two services are functionally similar. Since OWL-S considers only a single operation of a web service in a description, it needs modification to be able to represent multiple operations. This is currently handled by our system by modeling each operation as a separate service.

But the real benefit of that modeling multiple operations of a service would be if those operations can represent conversation with their clients (i.e. choreography of interaction with clients). Neither, OWL-S nor the templates of METEOR-S seems to fulfill this purpose, as of now. To summarize, ontological concepts are being matched for construction control flow and data flow in the logical composition phase. The output of this is an abstract BPEL which is abstract in the sense that it does not refer to invocable service instances but their types. Next, in the physical composition phase, this abstract BPEL is converted into executable BPEL. This is done by searching a corresponding web service instance, for each service type in the abstract BPEL, based upon their data typed portTypes and QoS parameters.

9.2 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 the matching of annotations, \[45\] formalizes matching of web services from a directory based on various inexactness measures. In \[6\], 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. While functional attributes have received attention, the non-functional attributes have not been much recognized in semantic web. They relate to performance, reliability and other user-acceptance issues. \[46\] describes how such requirements can be qualitatively arranged as goal structures and used to design systems. Their framework allows treating
requirements as potentially conflicting or synergistic goals to achieve during the software development process.

A general survey of planning based approaches for web services composition can be found in [47] which is applicable for our logical composition. SWORD [48] 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. A service is represented by a rule that expresses that given certain inputs, the service is capable of producing particular outputs. Web Services Modeling Ontology\(^{17}\) (WSMO) is a recent effort for modeling semantic web services in a markup language (WSML) and also defining a web service execution environment (WSMX) for it. Our logical composition approach is not specific to any particular modeling language and can adapt to newer languages.

Sirin et al. [49] 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 Section 5.

9.3 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 [50] have investigated dynamic service selection based on user requirements.

Zeng et al [25,51,24] describe two instance selection approaches, one based on local (type-level) selection of services, and the other based on global allocation of service types to instances using integer programming. They define

\(^{17}\) [http://wsmo.org](http://wsmo.org)
a generic QoS model and formulate the problem in a way that maximizes user satisfaction expressed as utility functions over QoS attributes, while satisfying the constraints set by the user and the by the structure of the composite service. They also discuss a replanning procedure which may be triggered under contingencies, e.g. a component service becomes unavailable or the QoS of one of the component services changes significantly, in order to ensure that the QoS of the composite service remains optimal.

In [22], the authors present a predictive QoS model that allows the workflow engine to estimate, monitor and control the QoS rendered to customers. Their model can be used in conjunction with our work, to effectively manage and estimate the QoS values of advertised service instances. The model also computes the QoS for workflows based on component task attributes. In this respect, this work can be seen as aggregating the QoS for a composite workflow from the individual instances used in composition. On the other hand, we try to solve the problem of picking instances for the composite web service with the aim of satisfying end-to-end QoS requirements. In [23], the author corrects some mathematical flaws in [22] and also extends this framework for the application of such considerations within web service discovery for workflow-defined applications. Ranking of instances based on QoS values has also been proposed by [22,23]. However, their ranking is for each service type whereas our ranking is based on end–to–end values of the QoS for the composite service.

[52] presents a system for dynamic service selection based on data mining techniques. The idea proposed consists of labeling process executions with quality measures. Other proposals such as METEOR [53] and CrossFlow [54] have considered QoS models for workflows along four dimensions namely time, cost, reliability and fidelity. Bonatti and Festa [55] consider optimal service selection for a given set of service requests (such as the activities occurring in a workflow) to a set of service offerings. They prove that one-time costs make the optimal selection problem computationally hard. In the absence of these costs the problem can be solved in polynomial time.

Our solution regarding generation of data-flow is related to [56] which describes an environment for building reusable ontologies based on the concept of roles. This work informally defines role as a characteristic that a basic domain concept exhibits in a context. We can use their tool to build role ontology in parallel with the domain ontology. An alternative proposal to OWL-S is the SESMA [57] model which directly handles inputs and outputs. Here, a notion of conversation data set is introduced to hold the input and output variables with values, and these could be evaluated as part of reasoning with the service’s preconditions and effects.

Finally, there has been a considerable effort in the Web service community
in identifying the challenges in workflow orchestration between component services. In [58], 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) [59]. 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.

10 Conclusion

We have described a two-step approach for end to end composition of web services by semantically annotating web service components, as well as a prototype that demonstrates this approach 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 results in significant reduction in the time-to-market for new services.

There are two key innovations in our solution. First, we decouple web service composition into logical and physical composition stages that address application 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 optimizing techniques in each stage that can adapt to changes in the service creation environment.

In the future, we plan to integrate the service composition system with a larger service development and execution infrastructure. For the Logical Composer, we 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 the Physical Composer, we will continue investigating different instance selection heuristics for QoS matching, that can accommodate constraints on different QoS metrics. Efforts are ongoing to enable a procedure to classify composition failure at different stages, and provide feedback on possible causes and remedies to overcome the failure. This would serve as a decision-support tool that can be used alongside the service composition system. Finally, we want to explore a feedback-based approach where the service creation system interacts with the runtime infrastructure to optimize the Quality of Service of the composite service,
based on changes in the execution environment.

11 Acknowledgments

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

References


