FUSION: A System Allowing Dynamic Web Service Composition and Automatic Execution

Debra VanderMeer
College of Computing
Georgia Inst. of Technology
deb@cc.gatech.edu

Anindya Datta
Chutney Technologyies
adatta@chutneytech.com

Kaushik Dutta
Chutney Technologies
kdutta@chutneytech.com

Helen Thomas
Heinz Sch. of Public Policy
Carnegie Mellon Univ.
hthomas@andrew.cmu.edu

Krithi Ramamritham
Dept. Computer Sci. and Eng.
Indian Inst. of Tech., Bombay
krithi@cse.iitb.ac.in

Shamkant B. Navathe
College of Computing
Georgia Inst. of Technology
sham@cc.gatech.edu

Abstract

Service portals are systems which expose a bundle of web services to the user, allowing the specification and subsequent execution of complex tasks defined over these individual services. Examples of situations where service portals would be valuable include making travel plans or purchasing a home. Service portals must be capable of converting an abstract user goal into a correct *and* optimal *concrete execution plan, executing according to the plan, verifying the result against a user's stated* satisfaction criteria*, and in the case of satisfaction failure, initiating the appropriate recovery procedures. The basic framework needed to support this functionality, from gathering the input to generating an optimal plan and executing that plan, is a prerequisite for all service portals, yet there are currently no such commercial systems in existence, and the research literature has given only cursory treatment to some of these issues. In this paper, we describe FUSION, a comprehensive software system which provides the underlying framework for a service portal. We show how using the elements of this framework, service portal designers and architects can easily create domain-specific service portals, e.g., a travel service portal. We also present the Web Services Execution Specification Language (WSESL), a language that we have developed to describe execution plans in the context of the FUSION services model. Finally, we develop a set of data structures and algorithms for generating correct and optimal execution plans.*

1 Introduction

The *Web Services* paradigm allows for the development of loosely coupled "islands" of distributed computing. Here, independent computing resources expose an interface of available operations, accessible over the network. For example, a bank might expose credit card processing functionality as a service, allowing, for example, a travel agency to charge a traveler's credit card for airline or rail bookings across enterprise network boundaries. This type of interaction is made possible by a set of standards that define how web services are described, discovered, and invoked.

The ability to access functionality over the network in a modular fashion leads to the potential for *composing web services*. Web service composition can enable valuable functionalities for the end user that may be difficult or impossible to achieve when a user must interact with "individual" web services. For instance, suppose a user wishes to attend an open air event on a particular day only if the forecast for that day does not call for inclement weather. Further suppose the existence of a weather service as well as a service allowing the purchase of tickets for events. It is of course possible for the user to contact these services individually and decide whether she should purchase a ticket and then act accordingly. Alternatively, if a system were available that exposed the services to her from a single point and possessed the ability to generate and execute a "composite" plan based on her requirements, such a system would clearly provide value over and above the total value provided by the individual web services. It should be noted that the example above is quite simplistic. In more complex situations, e.g., making travel plans, or buying a house (imagine the complex interaction between inspectors,

mortgage lenders, lawyers, etc.), the value provided by such a system is greatly magnified. We refer to these systems as *service portals*. Service portals allow users to mix and match services based on their needs. Essentially, such systems need to provide "soup-to-nuts" functionality to users in satisfying service based, "real-life" needs, as illustrated in the examples given above. The need for such systems is well-documented in the literature, particularly in the area of service-selection for web services composition [5, 6, 7], yet the work in describing actual systems which support the full range of functionality of a service portal are virtually absent from the literature. Effectively, a service portal must perform the following tasks: (1) collect the user's goal information; (2) map his goals into actual executable callouts, including parameter bindings, and generate a logical representation of his satisfaction criteria; (3) convert an abstract user goal into a *correct* and *optimal* execution plan over the service portal's constituent services; (4) map the execution plan to code, and manage execution; (5) upon receiving the execution result, verify that the user's requirements have been satisfied; (6) for an execution not meeting the user's satisfaction criteria, initiate appropriate recovery procedures; and (7) deliver the results to the user.

These activities comprise a common task-set that any service portal, regardless of application domain, must perform. In other words, much in the same way that application servers provide a common infrastructure for building web applications, there is a need for a common infrastructure to build web service portals. In this paper, we propose the architecture and system details of FUSION, a software infrastructure system providing the common infrastructure elements needed to support service portals.

The reader might inquire at this point as to how this work goes beyond what has already been reported on the issue of dynamic web services composition, which is a well recognized problem in the literature [2, 5, 6, 4, 7]. This work takes the approach of exposing a bundle of services and allowing the user to specify an execution plan by choosing services from this bundle. Examples of such systems include eFlow [3] and Self-Serv [1]. These systems provide users with a set execution specification primitives (e.g., sequencing, parallelizing, conditionals), which are usually presented to the user in a graphical form, allowing a user to specify a plan, which is then executed. While this approach is clearly a step in the right direction, significant enhancements are required, a few of which we enumerate below.

First, the systems mentioned above *require the user to specify procedurally* how the constituent services of a composite service should be invoked. It is unrealistic, indeed infeasible in many cases, to expect the user to be able to specify execution plans. For instance, consider a user who wishes to book a trip for his tenth wedding anniversary, and wishes to replicate his honeymoon. Specifically, he would like to fly to the same Italian city (Venice), and stay in the same hotel (the Ambassador), as he did ten years previously. This itinerary is conditional on obtaining a room in the desired hotel; if he cannot reserve a room in this hotel, he does not want to book the itinerary. An important observation is that there are multiple possible execution plans that could satisfy the user's requirements. For example, either the hotel booking or the flight booking can occur first, or, alternatively, both bookings can take place in parallel, by invoking the respective services simultaneously.

In this particular case, it turns out that the user is significantly more selective in his lodging preference (he explicitly requires a room at a specific hotel) than in his transportation preference (any airline would do). Therefore, the optimal plan might be to book the hotel first, and if that succeeds, to then book the flight to Italy. This is optimal both from a user-preference point of view (the success or failure of the Italy itinerary is dependent on obtaining a reservation in a specific hotel), as well as from a cost-of-cancellation point of view (cancellation penalties are typically much higher for airline tickets than for hotel reservations). Such reasoning, however, is complex, and cannot be expected from a naive user attempting to arrange this trip. Thus, a critical enhancement to existing dynamic service composition systems is the ability to *automatically generate an* **optimal** *execution plan* from the abstract requirements a user may specify, rather than relying on the user to specify the execution plan.

Second, once execution of a plan is complete, the service portal should be able to verify whether the results meet the user's satisfaction criteria. For instance, while a user may have requested the purchase of tickets to three Broadway shows as part of a planned trip to New York City, he may be willing to accept two reservations, even if the third show is sold out. In other words, the user's goal may often specify more "stringent" criteria than he is willing to accept. This is an important distinction from classical transaction semantics, where the typical "all or nothing" rules apply. In order to support this feature, the service portal system must meet two enabling conditions: (a) there should be a mechanism for the user to specify his "minimum requirements" for satisfaction, and (b) there should be a mechanism for verifying that such requirements are met.

Third, and finally, in the case of a service failing to exceed satisfaction thresholds, appropriate recovery efforts need to be initiated.

We are building an infrastructure software system called FUSION, that will allow service portal architects and designers to build portals possessing the above-mentioned functionality. In this paper, we present a comprehensive treatment of the FUSION service portal software system, both from an architectural perspective, as well as a systems perspective. Specifically, this work contributes: (1)

a description of a comprehensive architecture for the FUSION software system; (2) a model to describe a web service method; (3) a description of how a user's abstract specification can be mapped into method instances and *satisfaction criteria*; a language for describing execution plans, the *Web Services Execution Specification Language* (WSESL); and a formal description of the notions of *correctness* and *optimality* with respect to execution plans.

In summary, we regard this paper as presenting one of the earliest (if not the first) research works in the domain of building comprehensive systems enabling the creation of composite web services. Indeed, we are not aware of any other published piece of work or commercial system that enables "true" dynamic composition, while, simultaneously, enabling analysis of design time specification as well as verification of run-time results.

Due to space constraints, we are able to present only a subset of the full details of the FUSION system in this paper. Specifically, we describe the overall architecture, a detailed description of the user interface to FUSION, and our proposed web service model. The actual plan-building algorithm and its theoretical underpinnings, as well as the mechanisms for executing plans, verifying results, recovering if necessary, and presenting the results to the user, will be published in an extended version of this work. The remainder of the paper is organized as follows. Section 2 describes a running example, which we use to illustrate the different features of the FUSION system. We then describe the architecture of FUSION in Section 3. In Section 4, we describe the system's user interface in detail. Section 5 concludes this paper.

2 Web Services Example

In this section, we present a detailed example, based on the simple travel scenario presented in Section 1. Our traveler would like to recreate his honeymoon trip, visiting the same hotel in the same Italian city as in his original trip. If he cannot obtain a reservation for a room at this particular hotel, he does not want to book the Italy trip at all. To make the example more realistic, further suppose that our traveler specifies a set of travel preferences in addition to the Italian hotel. In particular, for his first preference itinerary, he indicates that any airline will suffice for traveling to and from Italy, but that he is interested in three particular tour excursions, and that he would like to go on at least one, but as many of these three excursions as are available. Figures 1[A] and 1[B] graphically depict two possible compositions of the component web services our travel service application may invoke in the process of fulfilling the traveler's request. Which execution ordering is better? This is dependent on the criteria used to evaluate the plan. One of the major goals of the FUSION system is to provide the underlying mechanisms required to support the evaluation of plans according to different criteria, and, given a goal criterion, to build an optimal plan for it.

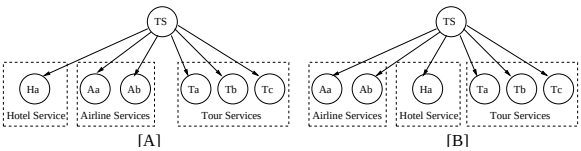

Figure 1. Example Travel Service Invocation

3 System Architecture of FUSION

As described previously, FUSION is a software system providing the base functional elements required of a service portal. Such a portal, given a user service specification, automatically generates a correct and optimized execution plan, then executes this plan and verifies the result. Figure 2 shows the FUSION system architecture. The system is divided into six subsystems: (1) *User Specification Subsystem* (USS), (2) *Web Services Dynamic Plan Generator Subsystem* (WGS), (3) *Plan Execution Subsystem* (PES), (4) *Verification Subsystem* (VS), (5) *Recovery Subsystem* (RS), and (6) *User Response Generation Subsystem* (URGS).

In this section we provide an overview of how the system works at a high level, by describing the functionality of each subsystem. (The detailed functioning of these subsystems is described in detail in subsequent sections.) As shown in Figure 2, the input to the system is a set of user requirements, which specify the service needs of the user and his satisfaction conditions. These inputs are fed into the User Specification Subsystem, described next.

3.1 User Specification Subsystem

The *User Specification Subsystem* (USS) is a graphical form-based interface that allows the user to (a) specify his abstract requirements, and (b) convert this abstract specification into a more structured form suitable for consumption by the downstream plan generator subsystem in the architecture. User specifications include the set of methods to be invoked and a logical expression representing the user's satisfaction conditions.

The input to the USS is an abstract goal and associated satisfaction conditions, specified through a set of forms. The output of the USS consists of two parts: (a) a set of parameterized method calls $T = \{W_i.m_p(), \ldots, W_n.m_s()\}$ where $W_i, \ldots, W_n$ are all web services accessible from the portal, and $W_i.m_p$ refers to the p^{th} method available from the published interface to W_i (the parameter bindings for these methods are omitted in the expression above); and

(b) the user's satisfaction specifications, represented with an *Execution Satisfaction Expression* (ESE) S. For the goal specified above, the user may wish to invoke `Airline` and `Hotel` services with the following respective methods: `getFlight` and `getRoom`. In this case, T = {`Airline.getFlight()`, `Hotel.getRoom()`}. Further assume that the user has specified the following success criteria: *both the hotel reservation AND the airline must be obtained*. In this case the satisfaction expression $S = (\texttt{airline.getFlight}() \wedge \texttt{hotel.getRoom}())$.

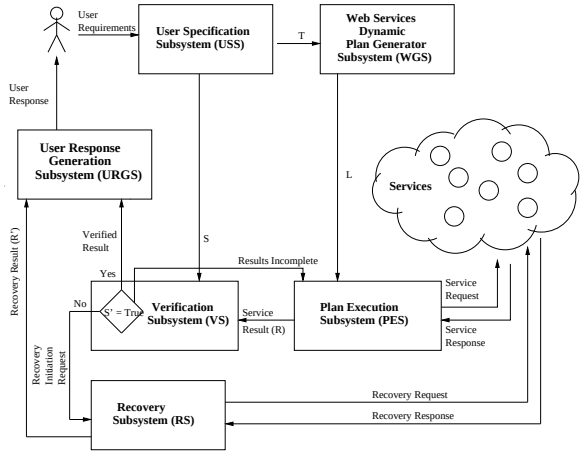

Figure 2. FUSION System Architecture

3.2 Dynamic Plan Generator Subsystem

The *Web Services Dynamic Plan Generator Subsystem* (WGS) takes as input the set of parameterized method calls T produced by the USS and generates a *correct* and *optimal* execution plan, denoted by L in Figure 2, which describes the "how" of executing the method calls in T. The plan is nothing but an expression stated in a language we propose called the *Web Services Execution Specification Language* (WSESL). A WSESL expression describes procedurally *how* a user's desired set of method instances are to be executed. Here, we first provide a few examples to give the reader an intuitive understanding of what we refer to as an execution plan. As shown in Figure 2, the WGS outputs an execution plan (L), which is passed to the *Plan Execution Subsystem*, described next.

3.3 Plan Execution Subsystem

The *Plan Execution Subsystem* (PES) takes as input the plan L generated by the WGS, evaluates L, maps it to executable code, and actually invokes the method instances described by L. The PES produces a result set R, which contains the set of service responses received from the completion of method processing, at each step of plan execution.

The PES interacts with the *Verification Subsystem* (VS) (described next) at each step of processing an execution plan. Here, as each web service method invocation completes processing, the PES sends the current result set for the execution plan to the VS, which checks the (possibly partial) results against the satisfaction expression S, to determine if (1) the expression is satisfied (i.e., S evaluates to `TRUE`); or (2) the expression cannot be satisfied, given the result set; or (3) the result set is insufficient to determine whether S is satisfied or not (i.e., further processing is required), denoted by the "Results Incomplete" interaction between VS and PES in Figure 2.

3.4 Verification Subsystem

The *Verification Subsystem* (VS) verifies that the service result meets the user specified satisfaction criteria. The VS receives two distinct inputs: (a) the service result R forwarded by the PES (as shown in Figure 2), and (b) the *Execution Satisfaction Expression* (ESE) S forwarded by the *User Specification Subsystem* (USS). Based on this dual input, the output of the VS is a binary relation (S', R), where S' is an attribute denoting the satisfaction or failure of satisfaction of the expressions S (or incompleteness of information for determining satisfaction of failure), and R is the service result received by the VS from the PES. S' is `TRUE` when the Satisfaction Checker in the VS detects no satisfaction condition violations, `FALSE` when the VS detects satisfaction condition violations, and `INCOMPLETE` when the result set in R provides insufficient information to determine whether S is satisfied or not.

3.5 Recovery Subsystem

If the value of S' is `FALSE`, the output of the VS is routed to the *Recovery Subsystem* (RS), which then initiates a recovery process, which rolls back (to the extent possible) the effects of the execution up to the point where S' is found to be `FALSE`. The results of these "undo" actions are combined with the result set R, to create a new result set R', and passed to the User Response Generator Subsystem for presentation to the user.

3.6 User Response Generation Subsystem

The verified result produced by the VS is input to the *User Response Generation Subsystem* (URGS), which is responsible for preparing and delivering the final response to the user. The output of the URGS is a user deliverable response, e.g., an HTML page. If any satisfaction condition violations were detected by the VS (i.e., S' evaluated to `FALSE`), then the appropriate error messages are included in the user response. The URGS receives this information

from the RS, in the result set R'. Otherwise, the user response contains the results of the optimal plan (e.g., a travel itinerary) drawn from R, obtained from the VS.

Having described the main components in the system, we now discuss our work with respect to these components.

4 The User Specification Subsystem

In this section, we describe the USS subsystem in greater detail. Recall from Section 3 that the USS converts a user's service requirements and satisfaction specification into a set of parameterized web service method calls T and an Execution Satisfaction Expression (ESE) S. This is accomplished through the three modules of the USS: the User Interface (UI), the Method Mapper (MM), and the Execution Satisfaction Expression Generator (EG). Figure 3 depicts these three modules graphically.

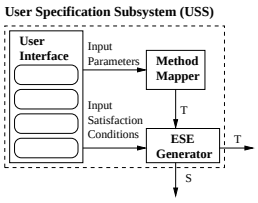

Figure 3. The User Specification Subsystem

4.1 The User Interface Component

The UI is primarily responsible for gathering user input. Here, the user is presented with a graphical form-based interface, which lists the available service options, and allows the user to specify *satisfaction constraints* over the services.

The UI also allows users to specify an element of the output of one service method as an input to another. For example, our traveler might wish to contract with an airport shuttle service to travel from his hotel to the airport for his return flight. Here, the service method for the `Shuttle` service may take as input the airline, flight number, and departure time of the user's outgoing flight, all of which are available as outputs from the `Airline` service.

4.2 The Method Mapper Component

Once the user has completed the service and parameter forms, the data collected from these forms is passed to the MM. The MM produces service method invocations based on the service and parameter specifications input by the user. For this, we will need a model of a web service, which we describe next.

4.2.1 A Web Services Model

The interface to a web service W is a set of methods $\mathcal{M}_W = \{M_1, M_2, \ldots, M_n\}$. We denote the j^{th} method of a web service W as $W.M_j$. Two distinct notions need to be modeled with regard to methods: (1) a *method schema*, and (2) a *method invocation instance*. The schema for a method $W.M_j$ is simply a two-tuple $\langle P_j, R_j \rangle$, where P_j is an ordered set of t input parameters $\{P_{j,1} \ldots P_{j,t}\}$ for method M_j, and R_j is the return object of M_j. We can fully specify the schema of a web service method as follows: $W.M_j(P_j, R_j)$. The schema for all methods in W is exposed through W's WSDL file.

Distinct from the notion of a schema is an actual invocation instance of a method. (Clearly, for a given method, there is always a unique schema, but many possible invocations). An invocation instance of a method models all the information that is necessary to execute that invocation as well as verify the result of the execution. Notationally, we will denote an invocation of method $W.M_j$ as $W.m_j$ (i.e., uppercase letters will apply to the method schema, while lowercase letters indicate an invocation instance), which is modeled as a three-tuple $\langle p_j, r_j, C_j \rangle$, where p_j is an instance of P_j, r_j is the return object resulting from the execution of $m_j(p_j)$, i.e., r_j is an instance of R_j, and C_j is a boolean-valued logical expression representing the user's success criteria for that specific invocation, i.e., a condition which must be met in order for the method invocation to be considered successful.

Using this notation, we fully specify a web service invocation as follows (for conciseness, we omit the relational operators and parameter values): $W_i.m_j(\{p_{j,1} \ldots p_{j,s}\}, r_j, C_j)$. Given the model described above, we can now describe precisely how a user's input is mapped to a set of method instances.

4.2.2 Mapping User Input to Method Instances

In order to map the user input to methods, we must first create an instance of each service method selected by the user based on the method schema, then fill in the parameters the user has specified for that method instance. These parameters are available in each method's WSDL description. The *constraintSet* is populated by the Execution Satisfaction Generator Component, which we describe next.

4.3 The Execution Satisfaction Expression Generator Component

The *Execution Satisfaction Expression Generator* (EG) takes as input T and the user "satisfaction criteria" input, maps those criteria to methods, producing an *Execution Satisfaction Expression* S, and populates the C_j member of each method instance m_i in T.

As noted above, a user may provide satisfaction conditions on both the service selection forms and parameter specification forms of the UI. Through these conditions, the user indicates the "acceptability" criteria for the results that would be returned from a service method invocation as well as the final plan execution. These conditions are of two types: (1) *intra-method conditions* and (2) *inter-method conditions*. An intra-method condition specification is a condition imposed on the return values of a specific method. An inter-method satisfaction specification involves the specification of an arbitrary logical expression which expresses a condition across multiple method invocations. The final expression S is a concatenation of all the intra-method and inter-method constraints, joined by the AND operator. It is passed to the *Verification Subsystem* (VS), where it is evaluated during the processing of the execution plan.

5 Conclusions and Future Work

In this paper, we have identified the requirements of a new breed of systems, which we refer to as *service portals*. We have described the FUSION service portal infrastructure system, which addresses these requirements by allowing portal designers to build portals that allow the design, implementation, and execution of arbitrarily specified multi-service plans. We have reported the architecture of FUSION and the web services model that FUSION implements. Future work will present fuller details of the FUSION system, including a description of how to actually build correct and optimal execution plans, how to execute such plans, verify the results, recover if necessary, and present the results to the end user. Further, future work will also report on the performance of the FUSION system.

References

[1] B. Benatallah, M. Dumas, Q. Sheng, and A. Ngu. Declarative composition and peer-to-peer provisioning of dynamic web services. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2002.

[2] A. Berfi eld, P. Chrysanthis, I. Tsamardinos, M. Pollack, and S. Banerjee. A scheme for integrating e-services in establishing virtual enterprises. In *Proceedings of the Workshop on Research Issues in Data Engineering (RIDE)*, 2002.

[3] F. Casati, S. Ilnicki, L.-J. Jin, and M.-C. Shan. An open, fl exible, and confi gurable system for service composition. In *Proceedings of the Second International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems*, Milpitas, CA, June 2000.

[4] M.-C. Fauvet, M. Dumas, B. Benatallah, and H.-Y. Paik. Peer-to-peer traced execution of composite services. In *Proceedings of the Second International Workshop on Technologies for E-Services (TES)*, pages 103–117, Rome, Italy, September 2001.

[5] D. Mennie and B. Pagurek. An architecture to support dynamic composition of service components. In *Proceedings of the Fifth International Workshop on Component-Oriented Programming (WCOP)*, Sophia Antipolis, France, June 2000.

[6] G. Piccinelli and L. Mokrushin. Dynamic e-service composition in dysco. In *Proceedings of the 21st International Conference on Distributed Computing Systems Workshops (ICDCSW)*, Mesa, AZ, April 2001.

[7] J. Yang, M. P. Papazoglou, and W.-J. van den Heuvel. Tackling the challenges of service composition in e-marketplaces. In *Proceedings of the 12th International Workshop on Research Issues in Data Engineering (RIDE)*, San Jose, CA, February 2002.