Verifying Constraints on Web Service Compositions

Zhengang Cheng, Munindar P. Singh, and Mladen A. Vouk
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-7535, USA
{zcheng, mpsingh, vouk}@eos.ncsu.edu

Abstract. Current service description and composition approaches consider simplistic method invocation. They do not accommodate ongoing interactions between service providers and consumers, nor do they support descriptions of legal protocols of interactions among them. We propose richer representations which enable us to capture more of the semantics of services than current approaches. Further, we develop algorithms by which potential problems in service compositions can be detected when services are configured, thereby leading to superior execution of composed services.

1 Introduction

Web services enable application development and integration over the Web by supporting program-to-program interactions. Relevant standards include Web Services Description Language (WSDL), Universal Description, Discovery and Integration (UDDI), and Simple Object Access Protocol (SOAP) [6, 18, 5]. These are intended, respectively, for describing, discovering, and invoking services. While current approaches represent much progress, they carry the baggage of traditional distributed objects approaches. Services are integrated through method invocation without regard to any higher-level constraints.

Current Web service techniques are limited to services where each operation is independent. A common example is stock-quote lookup, where each stock query is unrelated to every other query. But when we move from simplistic information lookup to interactive information search or to e-commerce, especially for complex business-to-business settings, it becomes obvious that current techniques are inadequate. In particular, we face two main challenges.

- What are useful ways of structuring the composition of services? Method invocation is appropriate for closed systems, but services are inherently autonomous and often to be used in long-lived interactions. For example, a long-lived interaction occurs in e-commerce when you try to change an order because of some unexpected conditions or try to get a refund for a faulty product. Even short-lived settings involve protocols, e.g., checking if the service requester is authenticated and properly authorized before accepting its order.

- What are appropriate semantic constraints on how services may participate in different compositions? As observed above, it may be required to compose services so as to carry out certain kinds of interactions. However, if a given service does not support such interactions, then the composition can fail at run time in unexpected ways. For example,

if an e-commerce service does not allow orders to be changed after a certain time has elapsed, it may be unsafe to use this service in certain settings, even if it is superior in other respects.

Current approaches, based on traditional closed systems, fall short of handling the above situations. WSDL allows us to capture the various methods but does not support constraints among those methods. Either too many methods will always be enabled or too few. However, WSDL's functionality is required to specify the methods supported by a service. Recently, Web Services Flow Language (WSFL) was proposed to describe compositions of services in the form of a workflow [14]. WSFL specifications tell us how different services ought to be invoked, e.g., in terms of ordering and parallelism among them, but they don't tell us whether a particular service that is bound in the workflow will in fact deliver the right interactions. XLANG [16] fits in Microsoft's BizTalk Server architecture. It describes the behavior of a single Web service. XLANG as it stands today provides a notation similar in spirit to workflow languages.

Our contribution is in capturing deeper constraints on what services are willing to offer, capturing richer requirements for service composition, and comparing the two to decide if a particular service is appropriate for the intended composition. Roughly, we think of WSDL as providing an underlying layer for our work. We enhance WSFL to define a service composition language that provides the necessary inputs to our reasoning approach. XLANG specifications are envisioned to drive automated protocol engines that will ensure that the specified message flows are obtained. Further, the intended direction is to define message exchanges among Web services. In this expanded form, XLANG will relate to our approach by providing elements of a service composition language.

Although the work reported here is still in an early stage of development, we believe it addresses important real-world concerns in the future expansion of the semantic Web. In particular, for the semantic Web technologies to penetrate real-life enterprise and scientific applications, the semantic Web will need as strong a representation of actions and processes as of data.

The rest of this paper is organized as follows. Section 2 introduces our representations and Section 3 applies them for verifying service compositions. Section 4 shows how our approach applies to service-level verification. Section 5 concludes with a discussion of the important themes, some of the relevant literature, and directions for further research.

2 Representing Composition Constraints

Because of their autonomy and heterogeneity, services are naturally associated with *agents*. Agents are long-lived, persistent computations that can perceive, reason, act, and communicate [10]. Agents act with varying levels of autonomy depending on environmental constraints and their previous commitments. Each agent provides a service and interacts through the exchange of messages, which denote business documents. We propose the Agent Service Description Language (ASDL) to describe the external behavior of agent services. An ASDL specification describes the messages understood by a service along with an interaction protocol it follows. Like WSDL, ASDL can be published and accessed through a registry. We propose the Agent Service Composition Language (ASCL) to describe the composition of new services from existing services. The imported services represent an agent role with its

behavior described in its ASDL description. An ASCL specification describes the interaction with other agent services.

2.1 Agent Service Description Language

ASDL enables us to define the behavioral characteristics of a Web Service. A Web service that implements behavior to preserve its autonomy is considered a Web agent service. In essence, ASDL is an behavioral extension to WSDL. It describes the constraints of service invocation to capture the external visible relationship between the operations.

The external behavior of the agent is demonstrated by its interaction with other agents. Such interactions occur through message exchanges. The internal implementation and reasoning logic is governed by the autonomy of the agent. For simplicity, we describe agent behavior via a finite state machine that models the allowed operation sequences.

The behavior of a service provider describes the allowed invocation sequence of operations. Invocations of this service must satisfy this sequence. We describe the agent behavior through a set of states and transitions between the states.

The state of an agent is used to maintain semantic constraints on actions. For example, a seller may require a buyer to log in before ordering some thing. Thus we can describe legal sequences of operation invocation. An operation state has a name for referencing and the operation it represents. For example, we can represent a state for the "order" operation defined in WSDL as follows.

<state name="order" operation="order"/>

ASDL contains states for all operations defined in WSDL with the name and operation attributes same as the operation name. We allow empty states which only have a name, but with no operation defined. They can be used to denote semantic states such as "ordered."

An interaction can proceed from one operation to another only if allowed by the modeled transitions. Each transition has a source operation and a destination operation. It may associate a condition to specify when the transition can occur. For example, the following specifies that the customer must successfully "Login" before "Ordering" any product. (The gating condition "expr" can involve terms from the messages exchanged in the protocol.)

<transition source="Login" target="Order" condition="expr"/ >

If there are no other transitions into "order," the customer must log in before ordering.

2.2 An E-Commerce Example

Figure 1 describes the behavior of an agent who requires login first, then enables querying and ordering products, leading to checkout and payment, and finally shipping.

We describe the seller's behavior with a behavioral extension of WSDL.

```
<behavior name="seller">
  <states>
    <state name="start" operation=""/>
    <state name="OQuery" operation="Query"/>
    <state name="Ordered" operation=""/>
  </states>
  <transitions>
    <transition source="Start" target="Register" condition="NULL"/>
```

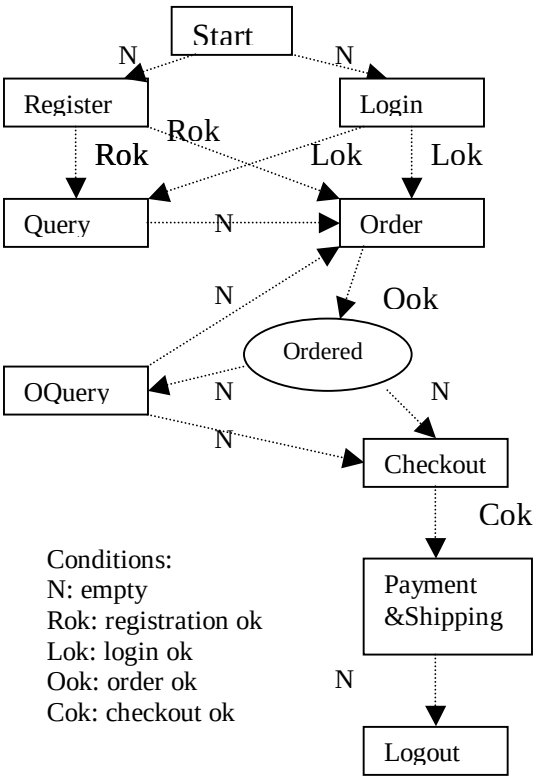

Figure 1: Behavior of a seller agent

```
    <transition source="Start" target="Login" condition="NULL"/>
    <transition source="Register" target="Query" condition="Rok"/>
    <transition source="Register" target="Order" condition="Rok"/>
    <transition source="Login" target="Query" condition="Lok"/>
    <transition source="Login" target="Order" condition="Lok"/>
    <transition source="Query" target="Order" condition="NULL"/>
    <transition source="Order" target="Ordered" condition="Rok"/>
    <transition source="Ordered" target="OQuery" condition="NULL"/>
    <transition source="Ordered" target="Checkout" condition="NULL"/>
    <transition source="OQuery" target="Checkout" condition="NULL"/>
    <transition source="OQuery" target="Order" condition="NULL"/>
    <transition source="Checkout" target="PaymentShip" condition="Cok"/>
    <transition source="PaymentShip" target="logout" condition="PoK"/>
  < /transitions>
< /behavior>
```

The following is a description of the transition conditions.

- NULL represents an empty condition.
- Rok, meaning registration OK, is a boolean expression based on the RegResult output message, and can be evaluated when the buyer receives the RegResult message.
- Lok, meaning Login OK, is a boolean expression based on the LoginResult output message.

- Ook, meaning Order OK, is a boolean expression based on the OrderResult output message. It indicates that the product is successfully ordered.
- Cok, meaning Checkout OK, indicates that an invoice is successfully send to the buyer.

The state *OQuery* represents that the buyer has ordered something and can check out at any time.

2.3 Agent Service Composition Language

Agent Service Composition Language (ASCL) specifications describe the logic of how a new service is composed from existing services. For reasons of space, we don't include details of ASCL syntax here. Suffice it to state that it enables services to be bound, and various flow primitives (sequencing, branching, and so on) used to specify desired compositions.

3 Verification of Service Composition

Using the above representations, we can determine whether service interaction protocols will be violated by the desired compositions. This can be done when services are bound—that is, at configuration rather than at run time.

We have developed some algorithms for reasoning based on the above representations. We lack the space to discuss these in detail. Briefly, these algorithms enable us to construct behavior state diagrams, find legal operations, and build an execution graph. We present the most interesting of these algorithms, which is for checking compliance of operation invocations.

To verify compliance at the operation level, we need the following data structures and functions.

- Build a service transition graph for each imported services.
- Find legal operations. Based on the service diagram, we can find the legal next states for a given current state.
- Remember history of invocation.

Given the above, following algorithm verifies a composition by traversing all possible execution paths.

1. Do a depth-first search on the resulted operation graph.
2. For each invocation of imported services, find the last state from the history, find the legal set of operation-states for the last state and current service from the corresponding service transition graph, check whether current operation is in the legal operation-state set. If not, there is an error. Otherwise add current operation to the history.
3. Continue until the whole operation graph is traversed.

Say a buyer finds the seller's service specification from the registry and would like to use the service to purchase a product from the seller. Figure 1 describes the seller's behavior in ASDL. It is up to the designer of the buyer agent on how to utilize the seller service. The verification algorithm is used for ensure that all possible execution paths in a service composition are legal. The following are some simplified examples to show how a service composition can be verified.

3.1 Sequential Composition

Suppose the buyer would like to implement the buy activity to buy products. (For brevity, unnecessary XML tags are not shown below.)

```
<consume role="seller" urlref="seller.xml"/>
  <activity name="buy">
    <sequence>
      <operation name="login" performedby="seller"/>
      <operation name="query" performedby="seller"/>
      <operation name="order" performedby="seller"/>
    </sequence>
  </activity>
```

Here the buyer invokes the login, query, and order operations of seller in sequence. Since login, query, and order are on the execution path, it is obvious that the above sequence is valid according to the seller's behavior.

3.2 Complex Composition

Now we consider the case where a buyer uses two seller services.

```
<consume role="S0" urlref="seller0.xml"/>
<consume role="S1" urlref="seller1.xml"/>
<activity name="buy">
  <operation name="register" performedby="S1"/>
  <fork condition="expr">
    <thread name="t1">
      <sequence>
        <operation name="login" performedby="S0"/>
        <if condition="expr">
          <sequence>
            <operation name="query" performedby="S1"/>
            <operation name="order" performedby="S0"/>
          </sequence>
        </if>
        <operation name="order" performedby="S0"/>
      </sequence>
    </thread>
    <thread name="t2">
      <sequence>
        <operation name="query" performedby="S0"/>
```

```
<switch variable="name">
  <case condition="expr2">
    <sequence>
      <operation name="order" performedby="S0"/>
      <operation name="query" performedby="S1"/>
    </sequence>
  </case>
  <case condition="expr3">
    <sequence>
      <operation name="query" performedby="S0"/>
      <operation name="checkout" performedby="S1"/>
    </sequence>
  </case>
</switch>
<operation name="order" performedby="S0"/>
</sequence>
</thread>
</fork>
</activity>
```

The above activity contains a *fork* construct. The threads in the *fork* construct are sequentialized. Based on the algorithm, we can build the execution graph as Figure 2. From Figure 2, we can find the following possible execution paths:

- Path 1: *S1:register, S0:login, S0:checkout, S0:query, S0:order, S0:query, S0:order*. Path 1 is not valid. The operation *S0:checkout*, is illegal since seller *S0* requires the buyer to *order* something before *checkout*.
- Path 2: *S1:register, S0:login, S1:query, S0:order, S0:checkout, S0:query, S0:order, S0:query, S0:order*. Path 2 is legal for both service provider *S0* and *S1*.
- Path 3: *S1:register, S0:login, S0:checkout, S0: query, S0: query, S1: checkout, S0:order*. Path 3 is illegal for the same reason as path 1.
- Path 4: *S1:register,S0:login, S1:query, S0:order, S0:checkout, S0: query, S0: query, S1: checkout, S0:order*. Path 4 is illegal, since the buyer has not ordered anything from seller S1, before trying to checkout.

4 Service Level Verification

The above discussion dealt with verifying service composition at the procedural level. Now we discuss verification at the service level. The service provider needs to verify that all possible invocation sequence of its exported services defined in its ASDL file are valid for all services it imported.

However, we have to note that it is not possible to enumerate all possible invocation sequences of a service. For instance, given a simple service S exposing two operations: *a* and *b*, the number of possible invocation sequences is infinite, e.g., *a*, *b*, *ab*, *aa*, *bb*, and so on. Here our verification is based on its exported service behavior defined in ASDL file. For example, the defined behavior is like *a–>b*. We need to verify that the sequence *a*, *b* is valid.

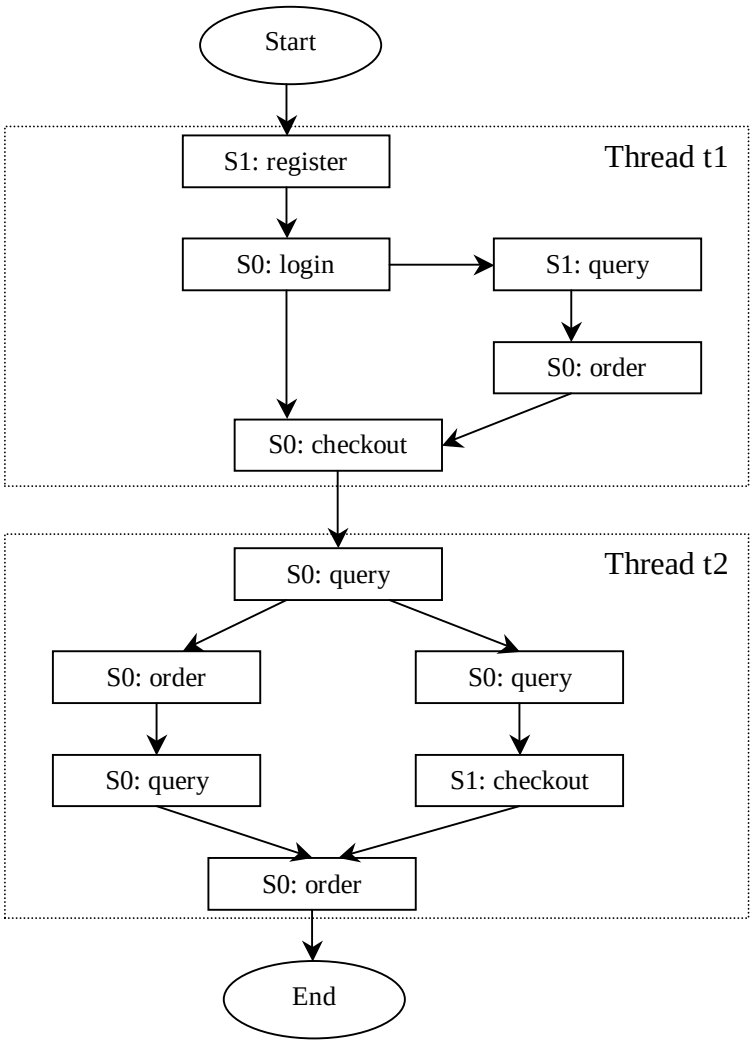

Figure 2: A composed service

Suppose there is a another operation *c*, which does not have an intrinsic relationship with *a* or *b*. We do not need to verify the validity of sequences involving *c*, since operation *c* is independent of operation *a* and *b*.

A service requester can be a final service consumer. It only use the service for its own purpose. Others are service integrators. They consume imported services and export a new service to the outside world. We need to find how verify at the service level.

For example, as in Figure 3, a service *C* is composed from imported services *A* and *B*. Service *C* has two exposed operations $c1$ and $c2$, and is then exposed as a new service to the outside world services like *D* and *F*. The behavior of service *C* is defined through a ASDL file, and published into a public registry. The ASDL definition can then be used to build new services manually or automatically. From the point view of Service *C*, it needs to ensure that invocation sequences following published service behavior in ASDL from outside like Service *D*, will not violate the restriction on the published behavior of services *A* and *B*. In service level verification, it needs to check whether all possible defined invocation sequences of operation $c1$ and $c2$ are valid.

Agents vary in their intelligence. Likewise, the services they provide can be simple or

complex. When a service consists only of behavior that can be captured by WSDL, the operations it supports are independent of each other. All invocation sequences of the stated methods are valid. For more complex services, some additional challenges must be addressed. Specifically, we consider the following complications during service level verification.

- Acyclic Directed Graph: The agent behavior can be described as a finite state machine. The verification can use algorithms similar to operation-level verification.
- Directed Graph with Loop: The agent behavior includes a loop. The verification algorithms can unravel the loop by using a modifier to limit the depth of the loop.
- Complex Directed Graph: The agent behavior can only be described by a complex graph. An example agent behavior could be like Figure 4, which consists of multiple loops in the definition. The service consists of four operations *m1, m2, m3, m4* with *S* and *E* designating the start and end state. How to verify a composed service against such complex services remains a challenge to be addressed.

At the service level, we can build a global execution graph by substituting the execution graph of each operation. For example, for a service *C* with exported operations *c1* and *c2* following the behavior *c1 –> c2*. The execution graph of operation *c1* is connected to the execution graph of operation *c2*, forming a global graph. Next, service compliance is checked against the execution graph.

This algorithm can handle acyclic directed graphs well, but it is not well-suited to complex graph that may include multiple loops as in Figure 4. In most practical situations, the service compositions are mostly coarse-grained with the invoked services doing substantial work. That is, the composition is relatively simple. The above algorithm should be enough for such situations.

4.1 Service Level Verification Example

Now we illustrate service level verification through the example of a travel service provided by a travel agent. The travel agent composes the services provided by hotel and airline agents. Its service is then exposed to customers. The customer may employ a customer agent to use the service.

We consider a simplified version of a travel agent. This only provides services such as query, purchase, and cancel. The travel agent requires the customer agent to login or register before purchase. A customer cannot cancel a ticket he has not yet bought. The airline and hotel agents require the travel agent to login, respectively, before booking a ticket or hotel room. They expose two operations *login* and *booking* and require the behavior that *login* should precede *booking*.

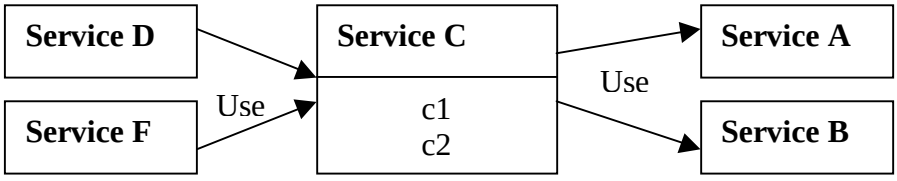

Figure 3: Service-level verification

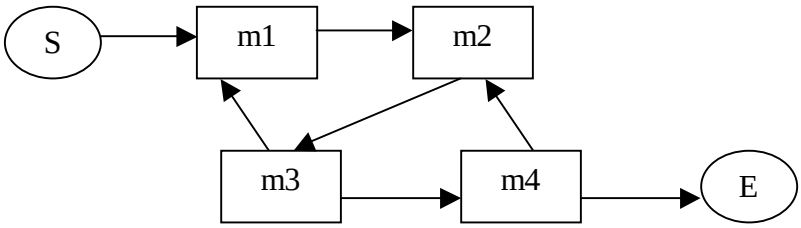

Figure 4: Complex service behavior of an agent

The travel agent exposes three operations, namely, *login*, *booking*, and *purchase*. Here, in operations *login* and *booking*, the travel agent will simply relay the request to the hotel and airlines corresponding to the *login* and *booking* operations. In the *purchase* operation, the customer will send both login credentials and a booking request to the travel agent; the travel agent will login to the airline and hotel agents, and book the ticket and room with them. The exposed behavior of these operations is that *login* should precede *booking*, while purchase is independent of the others. This scenario is illustrated in Figure 5. From the figure, it is easy to see that the service is valid for the composition.

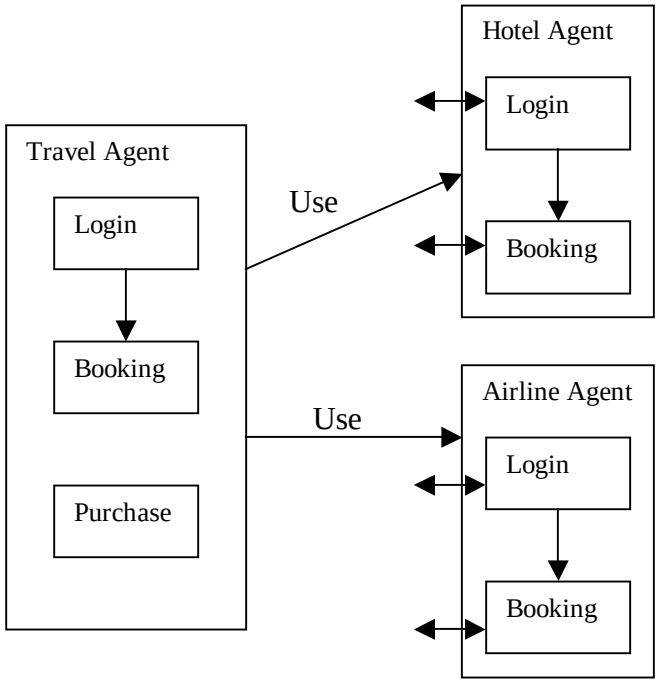

Figure 5: Travel agent scenario

Let us now consider an error situation. As shown in Figure 6, the travel agent exports *login* and *booking* operations. However, it tries to export a behavior wherein *booking* should precede *login*. In this case, the verification algorithm should detect the problem, because *login* should precede *booking* as required in its component hotel agent.

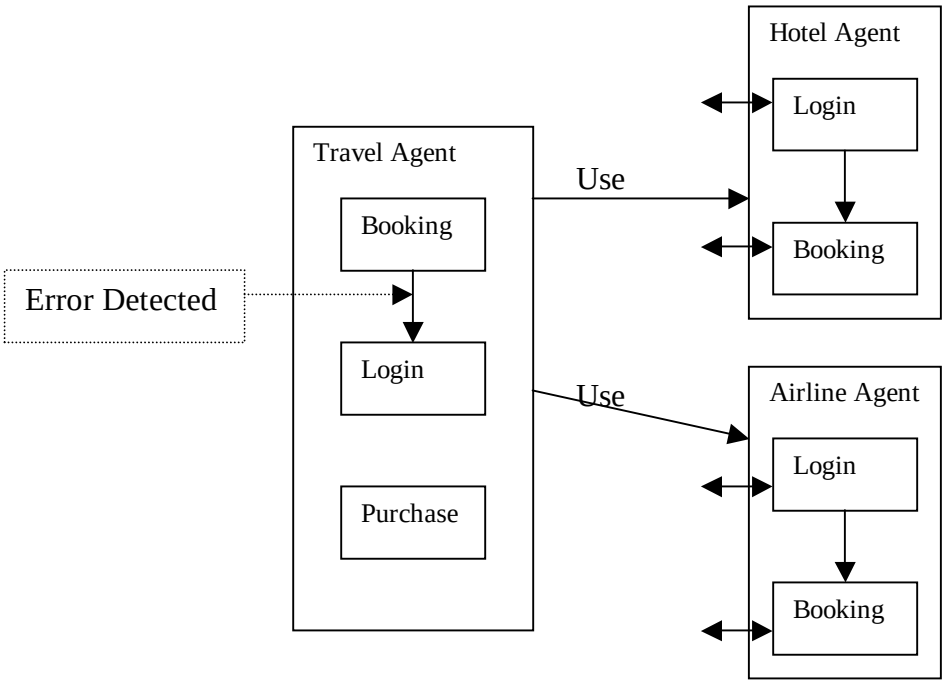

Figure 6: Service verification example: An error situation

5 Discussion

Although still in its infancy, the approach developed here seeks to facilitate the design and implementation of complex Web services as compositions of other Web services. Addressing this challenge is potentially of immense practical value. We believe it will be crucial to the expansion of semantic Web services into mainstream business process applications.

Our work treats Web service providers as autonomous agents, which can make independent decisions. ASDL exposes some behavior to the outside world. Our verification algorithms check the validity of a composed service, thereby detecting potential problems during the design phase of a composed service. We have developed a prototype tool to automatically check the validity of services.

5.1 Related Themes

Now we discuss some important topics concerning how our approach relates with service composition.

5.1.1 Automatic Service Composition

The above composition we considered is mainly at design time. For an autonomous agent, we might desire the agent to automatically come up with a plan of execution (a service composition based on other services). This remains a very difficult task. The DAML-S proposal provides a part of the foundation of automatic service composition, but much additional progress is necessary. DAML-S is a complex procedural language for web service composition. In particular, in order to achieve such a goal, the following problems must be addressed.

- Goal Definition: What states the composed service seeks to accomplish.

- Service Analysis: The agent should be able to analyze the goal and know what services might fulfill the goal.
- Service Selection: With the desired services known, the agent should be able the select individual services that can carry out the given task. This phase can be partially automated by using the classification of registries like UDDI and by some third party service rating services, e.g., [15].
- Execution Planning: To come up with a plan of execution.

Artificial intelligence tools such as Jess [7] may be applicable for reasoning, but defining a goal may still be too difficult in practice. Consequently, the user might be more inclined to compose the service himself.

Automatic Web service integration requires more complex functionality than SOAP, WSDL, and UDDI can provide. The functionality includes transactions, workflow, negotiation, management, and security. There are several efforts that aim at providing such functionality, for example, WSCL, WSFL, XLANG, BTP, and XAML. Entish [1] is also a relevant work in this direction.

5.1.2 Transactional Support

In real applications, transactional properties of services are important. It is reasonable that the service consumers or service providers may require two or more operations to be executed in as a transaction.

However, many business processes may need to run for a long time and it is not appropriate to run a whole process as a atomic transaction. Business processes can be modeled as long running processes, where the states and data can be stored, and be activated repeatedly over an extended period. To support such functionality, a messaging infrastructure, e.g., message queuing, should be in place between the service provider and consumer.

5.1.3 Error Handling

Error handling has not been adequately addressed in the context of composed services. In the real world, errors can occur anywhere between the service provider and service consumer. The service consumer should handle even possible networking errors. Service providers should expose the error messages in their ASDL. For example, the hotel agent might produce an insufficient funds error when attempting to charge the customer.

5.2 Literature

Besides Web services, the work described here touches upon extensive bodies of research on the semantic Web, workflow modeling, protocols, and agent-based techniques. We lack the space to review these in detail (but some were cited in the above discussion), but mention representative work here.

- Semantic Web. DARPA Agent Markup Language (DAML) [9] enables the creation of ontologies in the description of specific Web sites. DAML-S [2] is a Web service ontology from the semantic Web community [4]. DAML-S provides a core set of markup language

constructs for describing the properties and capabilities of Web services. Some emerging approaches add structure to service descriptions through the use of ontologies, e.g., [17].

- Workflow and process modeling. Klein & Bernstein develop a richer approach for describing and indexing services based on process models [12]. Verharen develops a contract specification language, CoLa, to specify transactions and contracts [19]. Verharen's approach captures obligations involving actions, but does not allow the obligations to be manipulated dynamically. This is a possible line of extension for the present work.

- Protocols. Barbuceanu and Fox [3] develop a language, COOL, for describing coordination among agents. Their approach is based on modeling conversations through FSMs, where the states denote the possible states a conversation can be in, and the transitions represent the flow of the conversation through message exchange. They try to handle exceptions through error recovery rules. HP's Conversation Definition Language [8] has similar goals to ASDL. CDL provides an XML schema for defining valid sequences of documents exchanged between Web services. Like ASDL, it uses conversations to model the externally visible interaction model of the Web service.

- Agent-based exception handling. Klein & Dellarocas exploit a knowledge base of generic exception detection, diagnosis, and resolution expertise [13]. Specialized agents are dedicated to exception handling. Our approach is complementary, since it applies at design time and does not require extensive intelligence.

5.3 Directions

This work opens up several interesting directions for research, some of which we are pursuing actively. On the practical side, we are working on our prototype to enhance its representational capabilities for services. On the theoretical side, it will be helpful to explicitly incorporate extended transactions in our models to capture richer constraints on service behavior.

6 Acknowledgements

This work was supported in part by the DOE SciDAC grant/contract DE-FC02-01ER25484 NSF grants CSS-9624425 and DST-0139037.

References

[1] Stanislaw Ambroszkiewicz and Tomasz Nowak. Agentspace as a middleware for service integration. In *Proceedings of Engineering Societies in the Agents World II*, pages 134–159, 2001.

[2] Anupriya Ankolekar, Mark Burstein, Jerry R. Hobbs, Ora Lassila, David L. Martin, Sheila A. McIlraith, Srini Narayanan, Massimo Paolucci, Terry Payne, Katia Sycara, and Honglei Zeng. DAML-S: Semantic markup for Web services. In *Proceedings of the International Semantic Web Working Symposium (SWWS)*, pages 411–430, July 2001.

[3] Mihai Barbuceanu and Mark S. Fox. COOL: A language for describing coordination in multi agent systems. In *Proceedings of the International Conference on Multiagent Systems*, pages 17–24, 1995.

[4] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic Web. *Scientific American*, 284(5):34–43, 2001.

[5] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (SOAP) 1.1, 2000. www.w3.org/TR/SOAP.

[6] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (WSDL) 1.1, 2001. www.w3.org/TR/wsdl.

[7] Ernest J. Friedman-Hill. Jess, the Java expert system shell, 1997. herzberg.ca.sandia.gov/jess.

[8] Kannan Govindarajan, Alan Karp, Harumi Kuno, Dorothea Beringer, and Arindam Banerji. Conversation definitions: Defining interfaces of Web services. http://www.w3.org/2001/03/WSWS-popa/paper20.

[9] James Hendler and Deborah L. McGuinness. DARPA agent markup language. *IEEE Intelligent Systems*, 15(6):72–73, 2001.

[10] Michael N. Huhns and Munindar P. Singh. Agents and multiagent systems: Themes, approaches, and challenges. In *[11]*, chapter 1, pages 1–23. 1998.

[11] Michael N. Huhns and Munindar P. Singh, editors. *Readings in Agents*. Morgan Kaufmann, San Francisco, 1998.

[12] Mark Klein and Abraham Bernstein. Searching for services on the semantic Web using process ontologies. In *Proceedings of the International Semantic Web Working Symposium (SWWS)*, pages 431–446, July 2001.

[13] Mark Klein and Chrysanthos Dellarocas. Exception handling in agent systems. In *Proceedings of the 3rd International Conference on Autonomous Agents*, pages 62–68, Seattle, 1999.

[14] Frank Leymann. Web services flow language. TR WSFL 1.0, IBM Software Group, May 2001.

[15] E. Michael Maximilien and Munindar P. Singh. Reputation and endorsement for Web services. *ACM SIGEcom Exchanges*, 3(1):24–31, 2002.

[16] Satish Thatte. XLANG, Web services for business process design, 2001. www.gotdotnet.com/team/xml-wsspecs/xlang-c/default.htm.

[17] David Trastour, Claudio Bartolini, and Javier Gonzalez-Castillo. A semantic Web approach to service description for matchmaking of services. In *Proceedings of the International Semantic Web Working Symposium (SWWS)*, pages 447–462, July 2001.

[18] UDDI technical white paper, 2000. www.uddi.org/pubs/Iru-UDDI-Technical-White-Paper.pdf.

[19] Egon M. Verharen. *A Language-Action Perspective on the Design of Cooperative Information Agents*. Catholic University, Tilburg, Holland, 1997.